TASKCTL

敏捷批量调度开拓者，开启批量调度工具化时代

敏捷调度技术平台

代码规则语法

成都塔斯克信息技术有限公司

产品网站：www.taskctl.com

1 前言......6
1.1 文档目的......6
1.2 读者对象......6
1.3 版本修订......6
2 作业控制器概述......7
2.1 基本概念......7
2.2 流程与调度核心的关系......8
3 流程结构......9
3.1 信息结构......9
核心信息......10
3.2 流程文件系统......10
模块文件......11
3.3 模块信息组织思路......11
3.3.1 传统设计思路......12
关系表达图形思路......12
数据表达思路......12
3.3.2 TASKCTL 设计思路......13
关系表达图形思路......13
数据表达思路......13
3.3.3 设计思路总结......15
4 流程总控文件......16

4.1 一份简单的流程总控文件例子......16

4.2 XML 关键字......17

4.3 流程基本信息......17

FLOWNAME-流程名称......17
DESC-流程描述......18
STARTMODUL-启动模块......18
ISSUBFLOW 是否为子流程......18
CTLBATCH 业务批次规则......18

4.4 模块信息......19

模块名称......19
模块描述......19

4.5 变量信息......19

变量名称......19
变量值......20
变量类型......20
是否加密......20

4.6 关于变量应用范围......20

提示：......21

5 流程模块代码......22

5.1 代码关键字......22

节点属性标签......24
组节点标签......24
系统缺省作业标签......24
自定义作业类型标签......25

5.2 模块代码基本特征......25

5.2.1 固定基本结构......25

5.2.2 属性继承与缺省......26

继承与缺省优先级问题......26
不是所有属性都存在继承与缺省特征......26
继承的有效范围......27

5.2.3 变量......28

变量定义......28
变量使用......28
流程缺省变量......29

5.2.4 关于代码排版特征的实际意义......30

5.3 模块代码设计......30

5.3.1 作业节点以及基本信息定义......31

NAME-节点名称......31
PROGNAME- 程序名称......31
PARA – 程序运行参数......32
EXPPARA – 程序运行环境参数......32
PREVSHELL / NEXTSHELL- 作业前/后置处理脚本......33
JOBDESC – 作业描述......33

5.3.2 调度控制策略......34

结构化控制策略......34
串并结构......35
SERIAL -串行......35
PARALLEL-并行......35
串并嵌套......36
循环结构......36
条件分支结构......37
互斥......38
TASKCTL 的 GOTO 语句 – LEAN(强制依赖)......39
执行计划控制策略......40
容错控制策略......42
循环控制策略......42
返回信息策略......43
分片作业策略......44
自动执行策略......44
优先级策略......44
超时失败策略......45
远程调度与负载均衡......45
利用 HOSTUSER 实现远程调度......46

自定义控制策略......46
虚拟资源控制策略......52
定时控制策略......53

5.3.3 流程结构化管理......54

INCLUDE - 模块引用......55
FLOW - 子流程调用......56

6 典型应用案例......57

6.1 流程每个批次的开始触发......57

时间触发......57
文件到达触发......58

6.2 流程翻牌处理......58

6.3 流程多模块设计......59

一个ETL 流程需求示例......59

6.4 子流程应用......62

设计子流程......62
编写特殊翻牌作业......62
调用定时器设计......62

6.5 回算流程......63

编写回算判断作业......63
配置回算流程......63
配置主流程......63

1 前言

1.1 文档目的

在调度控制应用中，控制器设计是调度实施中最重要的内容之一，这些设计内容我们可以统称为控制器信息。在一般调度方案或产品中，流程控制信息主要通过具有一系列的表结构进行组织，在实施时，主要通过以表单为主的界面或工具（比如 Excel）来实现流程的配置。但在 TASKCTL 产品中，控制信息主要通过具有一定规则的文本实现，且该文本信息直接面向用户设计开发。因此，掌握该文本信息的组织规则是设计 TASKCTL 控制器的根本前提。

本文旨在介绍 TASKCTL 控制信息的组织方式与相关规则，以及在具体调度中的应用。

1.2 读者对象

《TASKCTL 代码规则语法》主要适合以下读者对象：

- ✓ 技术开发人员
- ✓ 监控运维人员

1.3 版本修订

为了支持不同版本（4.1、5.0、5.1、6.0、7.0）的文档说明，在文档中会用如下标识来标注版本支持性：

v6.0＋：表示从 6.0 版本开始支持

v7.0－： 表示从 7.0 版本不再支持

从8.0版本开始，将采用web在线应用系统的图例来说明操作步骤。若需要查看Windows客户端的图例，请查看早期版本文档。

2 作业控制器概述

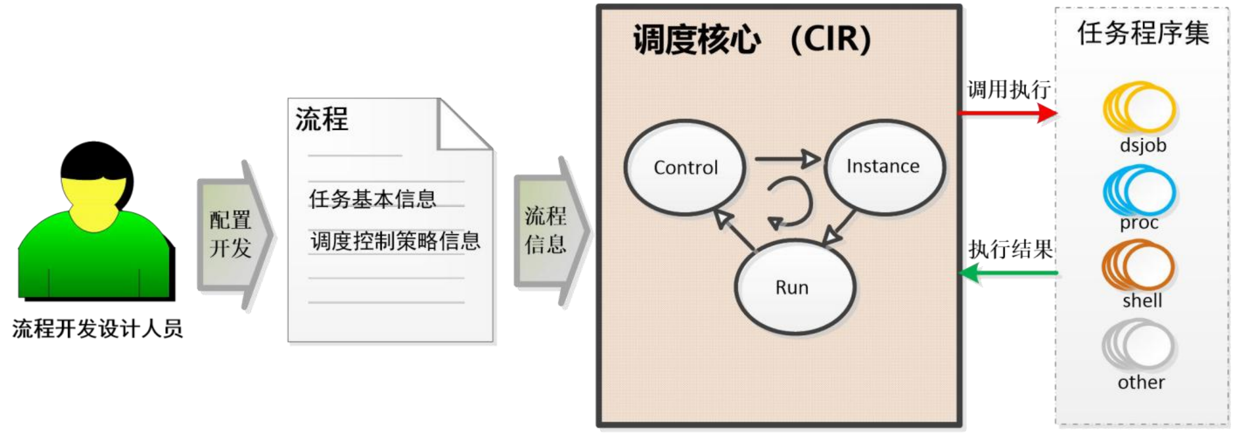
2.1 基本概念

理解作业控制器，我们首先需理解调度中与控制容器(流程、定时器)相关的几个基本概念，它们分别是：作业、控制器、流程、定时器和调度。

- ✓ **作业**：在系统后台处理中，我们经常把具有相关业务逻辑的处理由一个单独的执行代码、脚本、存储过程以及诸如 DataStage 第三方 ETL 工具开发的程序来完成，比如数据计算、文件拷贝、数据导入等。我们把这些由一个个程序完成的处理逻辑称为作业，有的资料将其称为作业。
- ✓ **控制器**：在目前调度业界，主要采用流程或定时的方式去调度、控制作业。TASKCTL 把这种流程或定时方式控制作业的容器统称为控制器。从技术本质来说，定时器的无序特征与流程控制器的有序特征形成了完整的控制技术体系。
- ✓ **流程**：在实际业务需求中，一个个单独的作业是不能满足需求的，他们之间可能是独立的，也可能存在一定的相互关系，比如运行先后关系等，同时也存在一定运行时间限制，我们需要把各个作业按一定的关系，在指定时间范围内运行才能满足需求，这种统一控制各个作业处理的过程称为流程控制。
- ✓ **定时器**：定时器是 TASKCTL 4.0 新增的技术概念。它的基本作用是按照一定的时间间隔定时地去控制执行无序的作业集合。在 TASKCTL 中表达流程和定时器相差不大。下面关于控制器信息结构、信息文件的组织结构以及核心信息模块代码文件的说明，主要以流程作为描述主体。
- ✓ **调度**：根据不同控制器的不同配置信息，将不同控制器的作业按要求执行过程就称为调度。

2.2 流程与调度核心的关系

TASKCTL 调度的核心作业就是按实际应用需求有序控制批量程序的运行，主要实施原理如下图所示：



批量调度工作原理

通过上图，我们可以知道，调度系统的控制目标是按要求执行各种各样的作业程序，为了完成这一目标，我们主要通过两部分工作完成。第一步，构建一个可以解析流程信息的控制核心，第二步，按项目需求开发配置流程控制信息。调度核心通过解析按照项目需求开发的流程信息，从而完成项目的调度需求。

由此可知，流程信息是整个调度的核心信息，对调度的实际运行起到决定性作用。

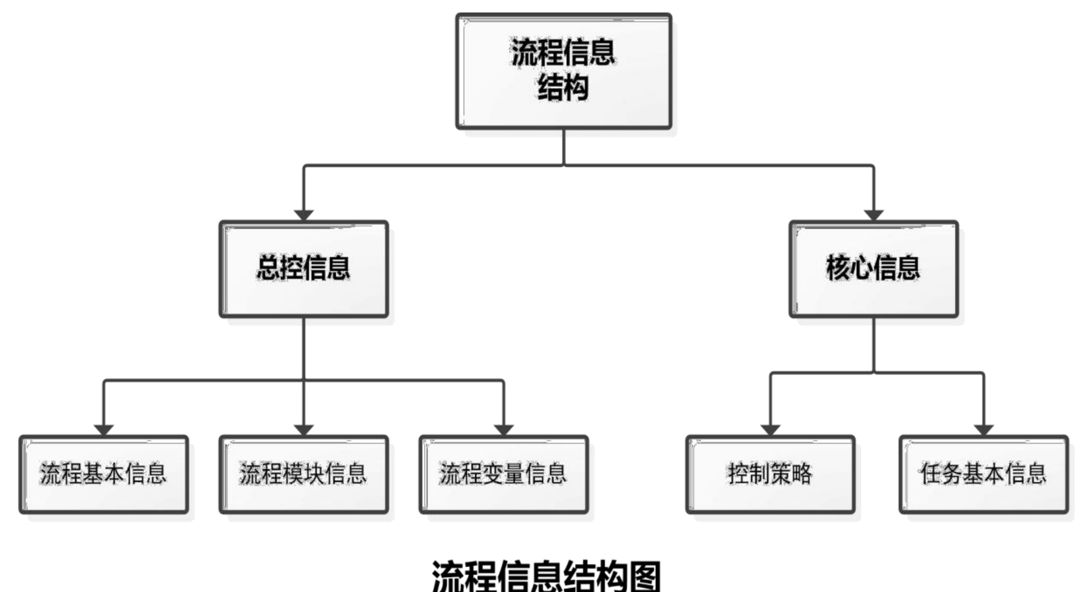
说明： CIR 指(Control Instance Run)缩写，是 TASKCTL 产品核心技术概念

3 流程结构

通过前面章节，我们分别了解控制器的基本概念与作用。本节主要说明控制器在实施时的主要信息结构、信息文件的组织结构以及核心信息模块文件的设计思路。

3.1 信息结构

为了完成调度，从信息层面上，TASKCTL 需要多种类型的信息对流程进行设计与描述，其信息结构如下：



流程信息结构图

由上图可知，流程信息主要包括两种类型的信息：总控信息与核心信息。

总控信息

总控信息指流程的主要概述信息以及一些基本控制信息，我们可以将其称为流程的工程信息，就像传统面向对象编程一样，在设计程序时，需要一个工程文件，通过该文件来描述程序的基本信息、基本引用、主程序、类信息等概述信息。

- ✓ **基本信息**：主要描述流程名称、流程类型以及启动模块等信息。
- ✓ **模块信息**：TASKCTL 流程的核心信息通过模块文件描述，在总控信息里描述该流程所包含各模块的清单信息。
- ✓ **流程变量**：TASKCTL 流程设计具有变量特征（以下相关章节会详细说

明），流程的各个私有变量在信息分类时，TASKCTL 将其归类为总控信息。

核心信息

流程核心信息是流程设计的主要内容，包括作业基本信息与控制策略信息。

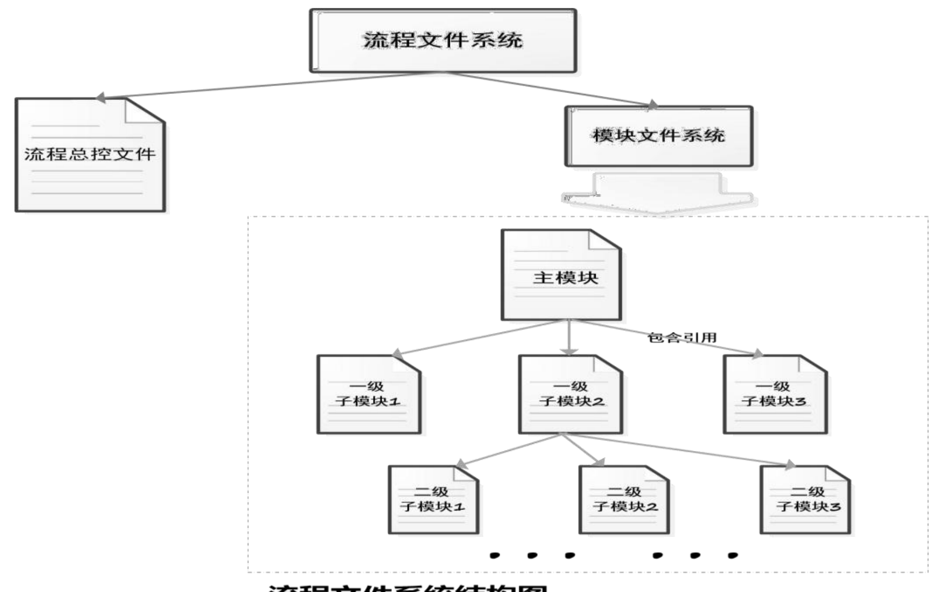
- ✓ **作业基本信息：**作业基本信息是流程的主要内容之一，它用于描述调度目标程序的各种信息，比如 Datastage 开发 Job、Informatica 开发的 session、存储过程、脚本程序等。作业基本信息主要包括：作业名称、作业对应的程序名称、作业运行所需的参数等。
- ✓ **控制策略信息：**控制策略信息是调度核心灵魂信息，该信息决定了调度平台在什么情况下、什么时候怎么调用作业目标程序。控制策略信息的本质为相关作业运行条件信息。比如依赖条件、互斥条件、执行计划条件等。

3.2 流程文件系统

上节描述了 TASKCTL 流程信息的基本结构，本节主要描述流程的信息主要组织结构。

TASKCTL 是无数据库调度平台，对于客户化的各种流程信息是通过具有一定组织特征的文件系统来存放，并通过 XML 语言描述各类信息。

TASKCTL 流程文件系统结构如下：



流程文件系统结构图

由上图得知，流程的文件系统主要包括两类文件：流程总控文件与模块文件。**总控文件**

总控文件，主要存放流程的总控信息。一个流程必须且只有一个总控文件。

模块文件

了解模块文件之前，首先需要弄明白什么是模块。

在实际 ETL 应用中，一个流程可能包含几百甚至上千个作业，使其作业基本信息与相关控制策略等信息非常庞大，为了有效管理这些信息，TASKCTL 引入模块概念，模块指具有一定关系的作业集合。对于大流程，用户可以根据作业的运行顺序、运行关系以及作业间的业务关系等信息，通过模块来划分并对流程进行重新组织，这不仅方便流程管理，同时也使流程变得更清晰，如下图所示：

在技术方面，一个模块通过一个文件来描述，且文件名称即模块名称。而一个流程包含一个或多个模块并有且只有一个主模块。主模块是流程的入口模块，就像程序的入口函数一样。对于流程主模块，由总控信息决定。

另外，由流程文件系统结构图得知，流程模块组织关系是以主模块为根节点树状结构关系，其中父子级联关系是通过父模块中相关节点定义确定的，具体参见模块代码相关章节。

3.3 模块信息组织思路

模块作为流程核心信息的基本组织单位，同时也是用户开发设计流程的主要对象。为了更好地设计流程，首先需要深入理解模块信息的组织思路。

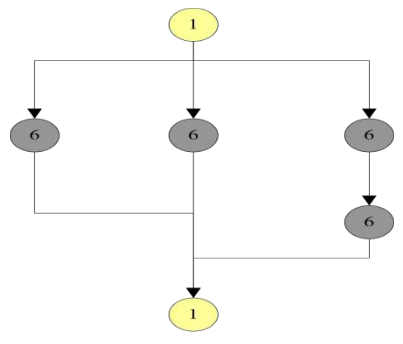
TASKCTL 流程以模块为单位的流程核心信息组织思路是有别传统的思路，是一种创新的思路。以下我们以与传统思路比较的方式对该思路进行描述。

3.3.1 传统设计思路

在调度业界，流程核心信息主要组织思路为：将作业节点化，并将作业控制策略属性化，其中依赖关系、并行关系是最主要的控制策略内容。

关系表达图形思路

在 ETL 调度界，流程图主要根据流程作业节点以及节点关系进行表达。图形表达如下：



非结构化示意图

由图可知，该图简洁且直观描述了各作业的依赖关系与并行关系。

数据表达思路

作业号	作业名名称	前驱作业	其它属性及策略
1	beginjob		
2	job2	beginjob	
3	job3	beginjob	
4	job4	beginjob	
5	job5	job4	
6	endjob	job2, job3, job5	

面向用户的设计方案

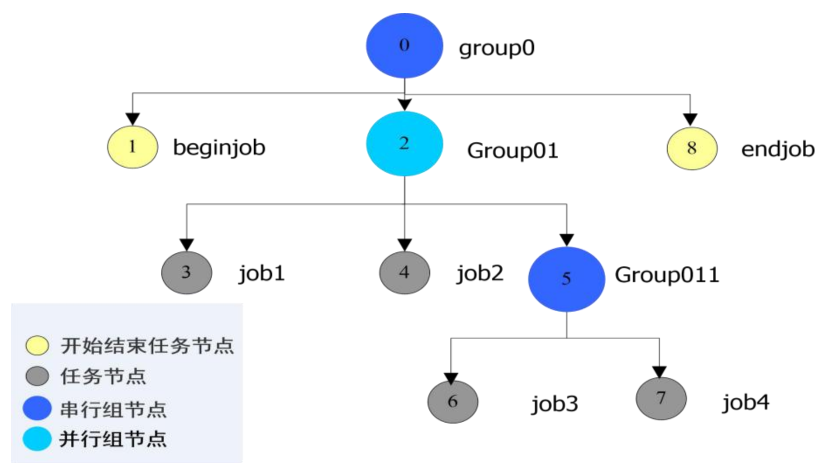
传统设计思路比较简洁直观，每个节点之间相对独立。在面向用户实施方面，主要是对每个作业节点直接以记录方式描述。具体实施手段，主要是图形拖拽以及节点表单方式描述。比如 Control-M 的作业对话框描述以及 Excel 方式描述。

3.3.2 TASKCTL 设计思路

关系表达图形思路

TASKCTL 设计思路与传统设计思路相比，共同点是将作业节点化，而不同点是对主要控制策略信息依赖并行关系表达方式的不同，TASKCTL 未采用对依赖以简单节点属性方式描述，而是站在整个流程的角度，采用串并节点方式进行结构化描述。该方式相对传统方式有一定的抽象。

表达思路如下图所示：



结构化示意图

该方式与传统方式相比，主要是增加了串并节点的思想，作业之间的运行关系不是主要通过依赖属性确定，而是通过上级组节点的串并属性决定。

数据表达思路

节点号	节点名称	节点类型	上级节点号	其它属性
0	group0	串	-1	
1	beginjob	作业	0	
2	group01	并	0	

3	job2	作业	2	
4	job3	作业	2	
5	group011	串	2	
6	job3	作业	5	
7	job4	作业	5	
8	endjob	作业	0	

面向用户的设计方案

由 TASKCTL 结构化示意图得知，流程整个组织思路结构化特征非常明显，其结构是具有唯一根节点的标准树结构。为此，TASKCTL 在流程核心信息组织时，直接采用面向用户并以 XML 语言为载体的文本进行描述。

以下是对前面 TASKCTL 结构化示意图的 XML 描述：

```
1  <serial>                          <!--流程最外层串行组-->
2    <name>Group0</name>
3    <begin>                         <!--开始任务-->
4      <name>begninjob</name>
5    </begin>
6  <parallel>
7    <name>Group1</name>             <!--并行组Group1-->
8    <job>                           <!--任务job1-->
9      <name>job1</name>
10   </job>
11   <job>                           <!--任务job2-->
12     <name>job2</name>
13   </job>
14   <serial>                        <!--串行组Group11-->
15     <name>Group11</name>
16    <job>                          <!--任务job3-->
17      <name>Job3</name>
18    </job>
19     <job>                         <!--任务job4-->
20      <name>Job4</name>
21    </job>
22   </serial>
23 </parallel>
24   <!--  用户模块代码自定义区开始 -->
25   <!--  用户模块代码自定义区结束 -->
26   <end>                           <!--结束任务-->
27     <name>MainModul_endjob</name>
28     <jobdesc>end</jobdesc>
29   </end>
30 </serial>
```

由以上信息可以简单看出，流程的模块信息是通过 XML 语言对图形直接描述，该信息具有明显的结构特征。由于是以文本的方式组织，使信息更为扁平化， 更易编辑。

实际 TASKCTL 方案中，为了使流程设计更容易、更简单，TASKCTL 提供了专业的流程集成设计开发环境，在该开发环境中，用户既可以通过文本代码方式设计，又可以图形拖拽方式设计。

3.3.3 设计思路总结

从设计思路的角度，传统方式比 TASKCTL 思路简单，更容易让人理解，TASKCTL 组织思路相对有一定的抽象，但由于采用了一定结构化特征，在面对大量作业面前，该思路在理解的基础上，使用会更灵活、更快捷方便，完成同样调度设计，信息总量也相对较少。

4 流程总控文件

流程总控文件是通过 XML 语言描述的流程概述总控信息文件。该文件与模块文件不一样，不直接面向用户，其信息主要通过相应的流程设计工具软件中相关命令与界面操作完成。

虽然流程总控文件不直接面向用户，但对该文件的具体了解，不仅可以加深对整个流程的理解，同时也会对流程具体设计带来更大的帮助。另外，在 TASKCTL 的发展计划中，今后该文件会公开且直接面向用户，使用户可以对该文件直接进行编辑设计，从而增加设计的灵活性。

4.1 一份简单的流程总控文件例子

```
<def-flow>
        <flowname>SampleFlow</flowname>
        <desc>一个流程范例</desc>
        <startmodule>MainModul</startmodule>
        <workdateparaname>workdate</workdateparaname>
        <maxjob>5</maxjob>
        <issubflow>N</issubflow>
        <def-module>
                <module>
                        <name>MainModul</name>
                </module>
                <module>
                        <name>modul1</name>
                </module>
                <module>
                        <name>modul2</name>
                </module>
                <module>
                        <name>modul3</name>
                </module>
                <module>
                        <name>modul4</name>
                </module>
        </def-module>
        <def-para>
                <para>
                        <name>sysdate</name>
                        <value>2012/06/17</value>
                        <paratype>comm</paratype>
                        <encry>N</encry>
                </para>
                <para>
                        <name>workdate</name>
                        <value>2012-06-18</value>
                        <paratype>date</paratype>
                        <encry>N</encry>
                </para>
        </def-para>
</def-flow>
```

通过以上范例得知，流程总控信息主要由三部分构成，它们包括：流程基本信息、模块信息、流程变量信息等。

4.2 XML 关键字

流程总控文件 XML 标签关键字如下：

分类	关键字	作用
根节点	def-flow	整个流程控制信息文件描述XML 根节点
基本 信息类	flowname	流程名称
	desc	流程描述
	startmodule	启动模块名称
	maxjob	流程最大作业并行度
	issubflow	是否为子流程:Y-子流程；N-非子流程
	ctlbatch	业务批次规则
模块信息	def-module	整个模块组描述主节点
	module	单个模块描述主节点
	name	模块名称（模块属性）
	desc	模块描述（模块属性）
变量信息	def-para	整个流程变量描述主节点
	para	流程单个变量描述主节点
	name	变量名称（变量属性）
	value	变量值（变量属性）
	paratype	变量类型: comm-普通变量；date-日期类型变量（变量属性）
	encry	是否加密: Y-加密；N-不加密（变量属性）

4.3 流程基本信息

流程基本信息主要包括：流程名称、流程描述、启动模块、流程最大作业并行度、是否为子流程等信息构成。

flowname-流程名称

流程名称是调度平台对流程的关键索引信息，使用时注意以下几点：

✓ 唯一性：流程名称相对调度服务器是唯一的，不能重复

- ✓ 长度：流程名称长度不能超过 20 个字符
- ✓ 输入限制：名称不能数字开头，不能包含特殊字符，如：！@|#*…等。

desc-流程描述

流程描述指流程的说明信息，该信息可以由任意字符组合。另外，流程描述虽然不是必输项，但通过客户端工具具体应用时，该信息具有特殊的展示意义。

- ✓ 长度：名称长度不能超过 100 个字符
- ✓ 输入限制：不能包含特殊字符，如：！@|#*…等。

startmodul-启动模块

一个流程可能由多个模块组成，用户必须指定启动模块，以表示流程的调度运行入口。这与一个程序必须定义主函数类似。

在 TASKCTL 平台中，启动模块即为流程主模块。

issubflow 是否为子流程

子流程是 TASKCTL 调度平台重要概念，它从信息内容上与普通流程没区别。唯一区别是：子流程能被其他普通流程调用。

TASKCTL 引入子流程概念的主要目的：一方面与模块一样，是为了有效结构化管理流程信息，另一方面，是为了达到更佳的调度控制效果。子流程与模块本质区别在于：模块是流程的组成部分，而子流程不是调用流程的组成部分；子流程有自己独立的私有变量空间，而模块与调用流程具有同样的变量空间。该区别就像实际程序一样，模块类似程序内部的一个类，而子流程是一个独立程序，可以通过其它程序调用。

对于子流程的具体应用意义，在本文相关章节会有具体举例说明。

在流程总控文件中，该属性 Y 代表子流程；N 代表普通流程。

ctlbatch 业务批次规则

流程从头到尾运行一次，称之为一个批次。通过设定流程批次命名规则，以便于更好的理解和划分业务流程的逻辑运行批次。在 TASKCTL 中，可以利用流

程的 ctlbatch 属性来实现个性化的批次命名规则。比如可以跟业务逻辑日期变量进行关联：

<ctlbatch>核心系统第$(work_date)批次流程</ctlbatch>

通过以上批次规则，可以直观的在运行数据信息中进行展示："核心系统第20200701 批次流程"

4.4 模块信息

在总控信息中，模块信息主要列举流程的所有模块概要信息，内容主要包括：模块名称、模块描述。

模块名称

模块名称是一个流程内模块的关键索引信息，使用时注意以下几点：

- ✓ 唯一性：一个流程内，模块名称是唯一的，不能重复
- ✓ 长度：模块名称长度不能超过 30 个字符
- ✓ 输入限制：名称不能数字开头，不能包含特殊字符，如：！@|#*...等。

模块描述

模块描述指模块的说明信息，该信息可以由任意字符组合。另外，模块描述虽然不是必输项，但通过客户端工具具体应用时，该信息具有特殊的展示意义。

- ✓ 长度：名称长度不能超过 100 个字符
- ✓ 输入限制：不能包含特殊字符，如：！@|#*...等。

4.5 变量信息

总控信息中定义的变量属于流程私有变量，主要应用于模块代码。变量信息主要包括：变量名称、变量值、变量类型、是否加密等信息。

变量名称

变量名称是一个流程内所有私有变量的关键索引信息，使用时注意以下几点：

- ✓ 唯一性：一个流程内，私有变量名称是唯一的，不能重复
- ✓ 长度：变量名称长度不能超过 50 个字符
- ✓ 输入限制：名称不能数字开头，不能包含特殊字符，如：！@|#*...等。

变量值

在定义私有变量时，必须确定初始变量值。在整个调度应用过程，该值可能会根据用户的行为进行修改。比如业务日期类变量。

- ✓ 长度：变量值长度不能超过 200 个字符

变量类型

TASKCTL 调度平台变量类型主要分三类：日期类(date)、普通类(comm)、常量（const）。

- ✓ **日期类：**日期类变量指与自然日期或业务日期相关的变量。一般情况下，此类变量在调度应用过程会随不同调度批次的变化而变化。
- ✓ **普通类：**普通类变量是相对日期类而言的。一般情况下，普通变量不会随不同调度批次的变化而变化，比如作业程序经常用的数据库用户、密码变量等。但是当环境变化后，有可能变量值会改动，如脚本路径。
- ✓ **常量类：**通常设定值后，就不能被更改了。比如一些名称标识等。

是否加密

流程变量可能会涉及到一些系统的敏感信息，比如用户密码，为了信息的安全性，TASKCTL 调度对变量增加的是否加密属性，对于加密的变量值，用户是不可见的。

在流程总控文件中，该属性 Y 代表加密；N 代表不加密。

4.6 关于变量应用范围

以上 4.5 节描述了流程的私有变量，实际上 TASKCTL 的一个流程除了使用自身定义的私有变量以外，流程还可以使用工程变量（v7.0+）和平台常量。与私有

变量主要区别在于变量应用范围不同：

私有变量：只能应用于流程自身。

工程变量：可以应用工程下的所有流程

平台常量：应用于整个平台下的所有流程

提示：

对于平台常量相关详细信息，超出本文的描述范围，更多信息请参阅《TASKCTL Admin-平台管理》文档相关章节。

5 流程模块代码

由前面章节得知，模块是流程核心信息的基本组织单位，其信息既是通过XML语言进行描述，同时，以XML语言描述的模块文本又是直接面向客户设计应用，因此，TASKCTL为了信息组织更灵活、更简单、更易懂，借鉴了相关程序语言的设计思想，使模块信息即有一定的规则，又有一定的语法特征。实际上，在TASKCTL各文档中，我们将模块信息称为模块代码。

本节主要讲解模块代码的主要特征、基本规则以及各种核心调度的代码实现。

在实际流程开发设计时，用户主要通过流程集成开发环境Designer实现，既可直接编辑代码，又可通过图形拖拽方式设计流程，但深入掌握模块代码是图形设计的基础。

5.1 代码关键字

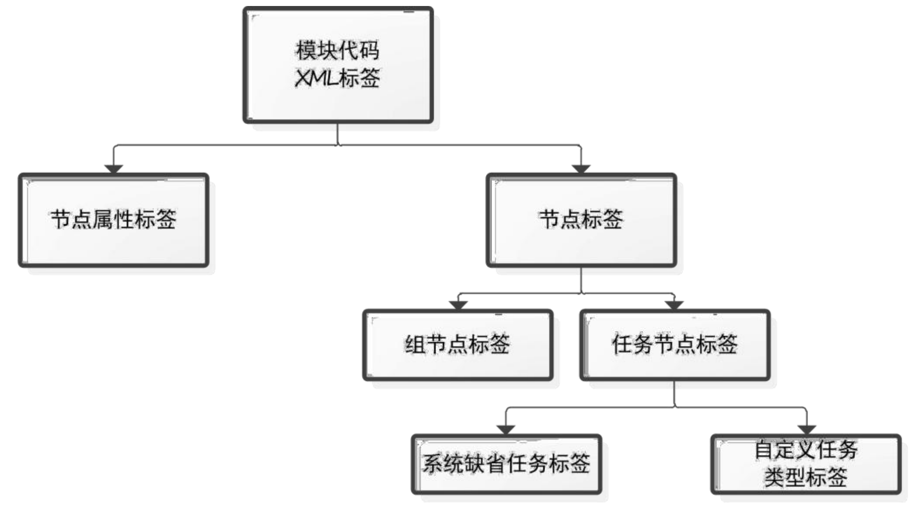
模块代码采用XML描述时，会涉及到很多XML标签关键字，用户对这些XML标签的了解，是流程模块代码设计的基本条件。

模块代码关键字如下表所示：

分类	标签名称	标签说明
组节点标签	parallel	并行组节点标签
	serial	串行组节点标签
作业节点标签-系统缺省类型	begin	模块开始节点标签
	end	模块结束节点标签
	nulljob	空作业标签
	include	模块引用节点标签
	flow	子流程调用标签
作业节点标签-用户自定义举例	selfmsg	自定义短消息作业标签
	filewatch	文件到达监控作业标签
	modivarv	修改变量作业标签
	sendevent	发送事件作业标签
	recvevent	接收事件作业标签
	ftpget	FTP 接收文件作业标签
	ftpput	FTP 发送文件作业标签
	sh	shell 类作业标签

	dsjob	datastage 类作业标签
	exe	可执行程序类作业标签
	oraproc	Oracle 存储过程作业标签
	python	Python 程序作业标签
	javaclass	Java 类作业标签
	javajar	Java jar 包作业标签
	ktrjob	Kettle 转换作业标签
	kjbjob	Kettle job 作业标签
	netezza	数据仓库数据库执行作业标签
节点属性标签	name	名称
	jobdesc	描述
	progname	程序代码名称
	para	作业程序运行入口参数
	exppara	作业程序运行环境参数
	monititle	监控标签
	datetype	日期类型
	period	时间周期
	maxnum	重复次数
	ignoreerr	错误或略错误
	agentid	执行代理节点
	hostuser	远程执行用户 v6.0+
	lean	依赖作业(组)
	ostr	互斥作业(组)
	cycle	循环次数 v7.0-
	cyclecount	循环次数 v7.0+
	cyclebreak	循环中断条件 v7.0+
	cycleinterval	循环间隔(s) v7.0+
	issplit	分片作业 v7.0+
	splitcount	分片个数 v7.0+
	autorun	自动运行 v7.0+
	priority	优先级 v6.0+
	timeout	超时失败 v7.0+
	prevshell	前置脚本 v7.0+
	nextshell	后置脚本 v7.0+
	condition	自定义条件
	SuccessV	执行成功返回信息定义
	ErrorV	执行错误返回信息定义
	FailedV	执行失败返回信息定义
	WarningV	执行后警告返回信息定义
	VirResource	虚拟资源消耗值
	TimingPlan	定时间隔(仅定时控制器内的作业可用)

上表列举了模块代码各种 XML 标签。标签主要分节点标签与节点属性标签两大类，节点标签又分组节点与作业节点，作业节点又分缺省作业与自定义作业，所有这些类型标签分类结构如下图所示：



模块代码关键字标签分类结构图

节点属性标签

节点属性标签主要用于描述 TASKCTL 流程中各种节点，比如基本属性以及一些控制属性。在实际应用中，各种属性对不同节点的有效性不一致，比如程序属性对组节点无效。具体有效性本文会在“模块代码设计章节“中分别体现。

组节点标签

组节点属于流程节点，主要包括串行节点与并行节点，是流程的基本控制节点。

系统缺省作业标签

系统缺省作业属于作业节点，是 TASKCTL 流程节点树状关系模型中的叶节点，代表一个作业，只是这种作业属于 TASKCTL 中内置逻辑作业，它们分别通过固定标签 flow、include、nulljob、begin、end 来表示。对这些缺省作业会在“模块代码设计章节” 中分别讲解。

自定义作业类型标签

自定义作业指 ETL 中用户开发的各种作业，比如 shell、datastage 类作业等。对此类作业标签关键字是由用户自定义确定。比如，对于 shell 脚本作业，我们既可以用’sh’表示，也可以用’shell’来表示。

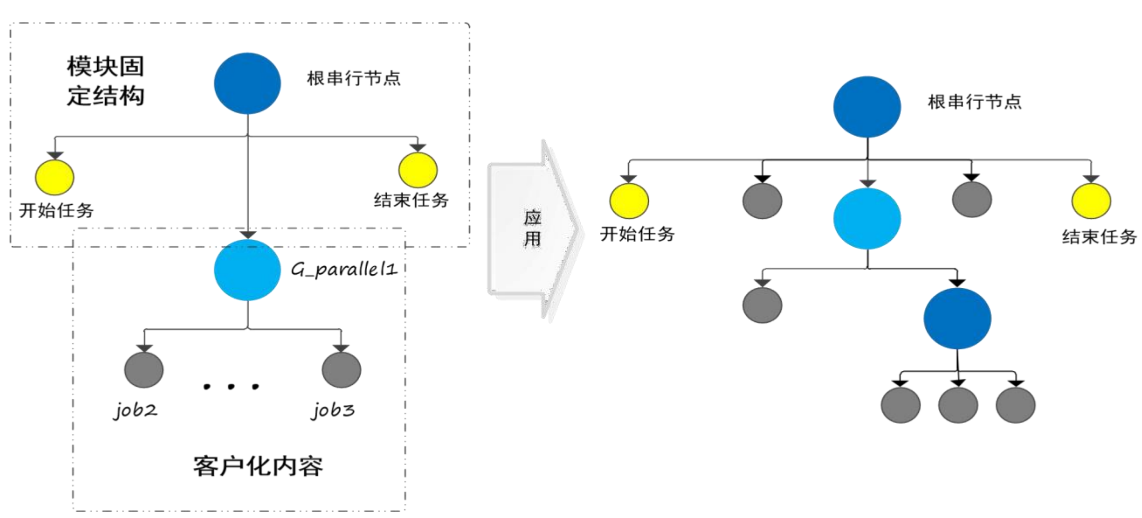
在实际应用中，自定义作业类型标签是通过管理平台定义的，具体方法请参见< TASKCTL 作业类型扩展应用>文档。TASKCTL 为了方便用户，也预设了一些常用自定义作业，如 ftpget、filewatch、selfmsg 等作业。

5.2 模块代码基本特征

模块代码是流程核心信息基本单位，它主要具备以下特征：

5.2.1 固定基本结构

为了统一模块的格式以及一些技术控制，TASKCTL 对模块采用统一的固定基本结构。该固定结构为：1、根节点始终为串行节点；2、二级节点第一个节点始终为开始节点；二级节点最后一个节点始终为结束节点。如下图所示：



这种不变的结构在代码中体现为：

```xml
<!--**************************************************************
    TASKCTL技术平台模块文件
      工程名称: project1
      流程名称: project1_Flow1
      模块名称: MainModul
 **************************************************************-->
<serial>
  <name>MainModul_rootnode</name>
  <begin>
    <name>MainModul_beginjob</name>
  </begin>

  <!--  用户模块代码自定义区开始 -->
  <!--  用户模块代码自定义区结束 -->
  <end>
    <name>MainModul_endjob</name>
  </end>
</serial>
```

模块根节点始终串行节点

第一任务开始节点

模块最后一个任务结束节点

5.2.2 属性继承与缺省

模块代码设计时，我们会设计大量的串并节点与作业节点，而每个节点都会存在很多属性，为了设计方便与代码简洁，TASKCTL 基于流程节点特殊的树状特征，引入了各种节点的属性继承与缺省机制。

- ✓ 属性继承：属性继承指下级节点继承上级节点的属性。下级节点只要不显示定义相关属性，下级节点自动继承上级节点的属性。
- ✓ 属性缺省：属性缺省指节点未显示定义某属性，又不能有效继承时，采用系统属性缺省值。

在实际应用中，对于继承与缺省我们必须要注意以下几点：

继承与缺省优先级问题

在模块节点树中，继承的优先级比缺省高。当一个节点上级节点相关属性不是缺省值，且下级节点没显示定义时，首先是继承，其次才是采用缺省值。

不是所有属性都存在继承与缺省特征

一个流程作业或组节点属性较多，但并不是所有属性都具有继承与缺省特征，比如节点 name 属性，该属性即不能继承也无缺省，用户必须显示定义；又比如

cycle 循环属性，该控制属性存在缺省，但不存在继承。

对于属性继承、缺省与具体属性关系如下表所示：

节点属性名称	节点属性描述	是否可以继承	是否可以缺省	缺省值
name	名称	否	否	
jobdesc	描述	否	是	空
progname	程序代码名称	否	是	空
para	作业程序运行入口参数	是	是	空
exppara	作业程序运行环境参数	是	是	空
monititle	监控标签	是	是	空
datetype	日期类型	是	是	msystime
period	时间周期	是	是	* * *
maxnum	重复次数	是	是	1
ignoreerr	是否忽略错误	是	是	N（不忽略）
agentid	执行代理节点	是	是	空（代表本地）
hostuser	远程执行用户	是	是	空
lean	依赖作业(组)	是	是	空
ostr	互斥作业(组)	是	是	空
cyclecount	循环次数	否	是	1
cyclebreak	循环中断条件	否	是	
cycleinterval	循环间隔	否	是	0
issplit	是否分片作业	否	是	否
splitcount	分片个数	否	是	空
autorun	自动运行	否	是	是
priority	优先级	是	是	10
timeout	超时失败	否	是	0
prevshell	前置脚本	否	是	
nextshell	后置脚本	否	是	
condition	额外自定义条件	是	是	空
SuccessV	执行成功返回值定义	是	是	0
ErrorV	执行错误返回值定义	是	是	1-98
FailedV	执行失败返回值定义	是	是	100
WarningV	执行后警告返回值定义	是	是	99
VirResource	虚拟资源消耗值	是	是	10
TimingPlan	定时间隔(仅定时器可用)	是	是	空(不调用)

继承的有效范围

子级模块能继承父级模块的属性，子流程不能继承父级流程的属性。

5.2.3 变量

变量是流程模块代码的基本特征，它与传统程序设计中的变量概念有一定区别，TASKCTL 中流程变量的本质是宏替换，目的是为了增加流程代码设计的可移植性与可维护性。以下以一个简单例子说明：

比如我们定义一个 shell 类型的作业节点时，需要设置 progname 程序名称属性，对于 shell 程序是需要确定程序路径的，但流程在不同环境中的部署目录可能不一样，为了保证流程模块代码的可移植性，我们就将脚本存放目录设置为变量， 对于不同的安装环境我们只需重新修改流程的相关变量即可。以下是一个变量应用实例：

```
12  <sh>
13    <name>testshell</name>
14    <progname>$(shellpath)/testshell.sh</progname>
15  </sh>
```

在流程运行时，该变量使用具体变量值

变量定义

对于流程变量的定义，变量不在模块代码中定义，而是在流程总控文件中定义，具体参见本文“4.5 变量信息“相关章节。

变量使用

在模块代码中我们用特殊格式表示变量，即用’$’加括号的方式表达，如下所示：

```
12  <sh>
13    <name>testshell</name>
14    <progname>$(shellpath)/testshell.sh</progname>
15    <para>-passwd $(password) -user $(user)</para>
16  </sh>
17  <sh>
18    <name>MainModul_JobNode0</name>
19  </sh>
```

变量:shellpath、password、user

通过以上相关章节，我们知道变量的本质是流程运行时的宏替换，在整个模块代码设计中，变量并不是应用于所有属性，让所有属性均能实现宏替换，能够使用变量的属性主要包括以下五种属性：

- ✓ progname：程序名称
- ✓ prevshell：前置脚本
- ✓ nextshell：后置脚本
- ✓ para：作业程序运行参数
- ✓ exppara：作业程序运行环境参数
- ✓ agentid：作业程序代理节点名称
- ✓ hostuser：远程执行用户
- ✓ condition：自定义控制策略
- ✓ ignoreerr：错误忽略条件
- ✓ splitcount：分片作业个数
- ✓ cyclebreak：循环中断条件

流程缺省变量

模块代码除了可以使用自身私有变量以及平台常量以外，还可以使用系统缺省变量。缺省变量主要包括：

- ✓ cycle：当前循环值，循环值从 1 开始。
- ✓ ctlid：流程 ID，在实际应用中，调度平台会自动给每个流程分配一个 ID 号
- ✓ renum：重做次数，对于一些错误作业，平台会不断重调，第一次运行前为 0，第一次运行完后为 1，当错误后第二次运行前为 1，当错误后第二次运行该值为 2，以此类推。

提示： 平台常量在 Admin 平台管理中统一定义。

5.2.4 关于代码排版特征的实际意义

模块代码主要是对流程树状节点的直观描述，并采用 XML 语言描述，为了使代码清晰直观，且使代码能从”形状”上能一定程度反应各节点上下级关系，我们对模块代码采用特殊排版：

- ✓ XML 语言标签节点竖排表示一个流程作业或串并组节点
- ✓ XML 语言标签节点横排表示作业或串并组节点属性
- ✓ 上下级节点间以及节点与节点属性之间采用缩进排列

提示： 特殊排版不是必须的，但为了使代码更具有直观性，我们强烈建议用户在具体实施时遵照以上排版规则。

下图表达模块代码特殊排列带来的直观性：

```
7   <serial>
8     <name>SubModul0_rootnode</name>
9     <begin>
10      <name>SubModul0_beginjob</name>
11    </begin>
12    <parallel>
13      <name>SubModul0_ParallelNode0</name>
14      <sh>
15        <name>SubModul0_JobNode0</name>
16       <progname>ddd</progname>
17      </sh>
18      <job>
19        <name>SubModul0_JobNode2</name>
20        <progname>ddd</progname>
21      </job>
22    </parallel>
23
24    <!-- 用户模块代码自定义区开始 -->
25    <!-- 用户模块代码自定义区结束 -->
26    <end>
27      <name>SubModul0_endjob</name>
28    </end>
29  </serial>
```

竖排serial标签表示一个串行节点
横排name标签设置该串行节点的属性

整体缩进排列分别表示了上下级节点以及节点
与属性之间的关系

5.3 模块代码设计

通过前面相关章节，我们了解了流程模块代码的基本特征以及相关基础知识，本节着重介绍模块代码具体实现。

5.3.1 作业节点以及基本信息定义

作业定义是模块代码设计的基本内容。通过作业定义，主要确定调度的目标对象是什么。

定义作业节点首先定义作业类型，其次是作业节点基本属性。 以

下举例分别定义 shell 与 dsjob(datastage 类型)两种类型的作业：

```
12  <sh>
13    <name>SubModul0_JobNode2</name>
14    其它属性定义。。。
15  </sh>
16  <dsjob>
17    <name>SubModul0_JobNode2</name>
18    其它属性定义。。。
19  </dsjob>
```

定义节点类型很简单，主要通过作业类型标签即可。在定义作业时，我们主要关心节点基本属性的定义与设置。

节点基本属性主要包括：

- ✓ name：节点名称
- ✓ progname：作业程序名称
- ✓ para：作业程序运行入口参数
- ✓ exppara：作业程序运行环境参数
- ✓ jobdesc：作业描述
- ✓ prevshell / nextshell：作业前/后置处理脚本

name-节点名称

每个作业节点（包括串并节点）必须输入名称，且对整个流程来说，不同模块中节点必须唯一，不能重复，名称输入需要注意以下几点：

- ✓ 唯一性：节点名称相对整个流程是唯一的，不能重复
- ✓ 长度：节点名称长度不能超过 60 个字符
- ✓ 输入限制：名称不能数字开头，不能包含特殊字符，如：！@|#*...等。
- ✓ 当作业类型为 include 时，name 代表模块名称

progname- 程序名称

对于自定义作业来说，程序名称指作业对应的实体程序名称，比如 datastage 作业的 Job 名称、Informatica 对应的作业程序名称、shell 程序名称、存储过程名称等。

对于非自定义作业来说，其中 begin、end、nulljob、include 特殊作业节点不需要该属性。而 flow 特殊作业节点，程序名称代表所调度的子流程名称，其用法在“流程结构化管理”章节会详细讲解，在此不做过多说明。

para – 程序运行参数

程序运行参数指作业程序所需的入口参数，比如 shell 的行命令入口参数，datastage 运行时 Job 的入口参数等。

对于不同作业类型的参数，TASKCTL 模块代码中统一用字符串表示。这个格式对于像命令行程序(shell、可执行程序、java 等)很好理解。比如 shell 程序，实际运行输入格式为：

“sh /home/myshellpath/test.sh -user abc –password cba;”

其中“-user abc –password cba”作为作业参数。

但对于一些需要特殊平台运行的作业，比如 datastage 程序，程序本身需要的参数格式是我们不可见的，对于这种作业，参数是通过对应插件按一定格式转换确定，具体参见《作业类型扩展应用》文档。

以下是存储过程与 datastage 作业参数应用：

```
12 <dsjob>
13   <name>SubModul0_JobNode2</name>
14  <progname>DSJob1</progname>
15  <para>user = $user:password = $password</para>      → datastage任务参数
16 </dsjob>
17 <proc>
18   <name>SubModul0_JobNode2</name>
19   <progname>DSJob1</progname>
20   <para>workdate=$workdate</para>    → 存储过程任务参数
21 </proc>
```

exppara – 程序运行环境参数

首先，我们要了解什么是环境参数以及环境参数与程序运行参数有什么区别。环境参数是指作业程序运行时，程序本身必备参数信息，比如 datastage 作业，

运行时，需要确定工程名称，为了可移植性，我们将该工程名称作为环境参数；又比如存储过程，运行时需要该存储过程的数据库名称、用户、密码等信息，我们就将数据库名称、用户、密码等信息作为环境参数。当然，有些作业类型不需要环境参数，比如 shell，该类程序直接运行即可。

环境参数与运行参数的区别在于：环境参数是相应程序在某种运行时，运行平台所需的参数信息；而运行参数只是程序内部需要的参数信息。以下是存储过程与 datastage 作业的环境参数应用例子：

```
13  <proc>
14    <name>shelljob</name>
15    <progname>proc_job1</progname>
16    <para>workdate=$workdate</para>
17    <exppara>user=$(user),password=$(password)</exppara>  → 存储过程环境参数
18  </proc>
19  <dsjob>
20    <name>dsjob1/name>
21   <progname>ds_job1</progname>
22   <para>user=$(user),password=$(password),workdate=$(workdate)</para>
23   <exppara>projectname=ocrm</exppara>  → datastage任务环境参数
24  </dsjob>
```

TASKCTL 将运行参数与环境参数这两种不同信息概念化并区别对待，不仅使应用更灵活，而且还使应用接口更清晰。

prevshell / nextshell- 作业前/后置处理脚本

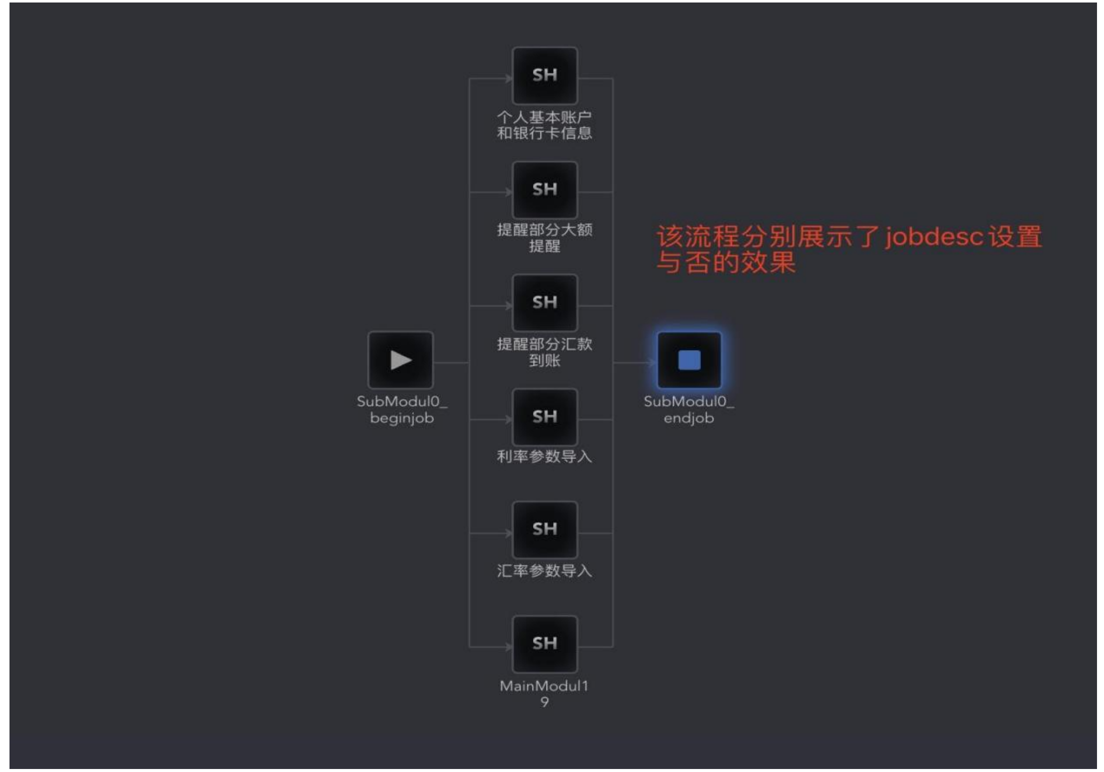
可以实现一些与业务流无关的操作，比如创建目录，导入环境等。

```
13  <perl>
14    <name>SubModul1_JobNode2</name>
15    <prevshell>$HOME/clearlog.sh</prevshell>
16    <nextshell>mv $HOME/log/*.txt $HOME/log/bak/*.txt</nextshell>
17  </perl>
```

jobdesc – 作业描述

作业描述属性指作业的详细说明，使用户通过该属性可以了解更多的作业相关业务等信息。

在实际应用中，该属性不仅仅是对作业简单描述而描述，同时还具备特殊的用户体验效果。对于设置了作业描述的作业，通过客户端工具展示，效果如下：



由图可知，对于设置了作业描述属性的作业，平台工具软件可以直接展示汉字说明信息，这使流程图变得更直观、更易理解。

5.3.2 调度控制策略

通过作业定义，完成了在模块代码中作业的描述。接下来，我们需要设计怎么去调用这些作业，即需要在模块代码中设计各种所需的调度控制策略。只有通过灵活设计各种调度策略，才能完成各种复杂多样的调度控制。

结构化控制策略

TASKCTL 结构化控制策略主要借鉴了传统应用程序开发的一些结构化思想，它不仅使流程设计更简单，也更灵活多变。同时，该结构化理念也充分体现了 TASKCTL 流程设计有别与传统调度流程设计的表单配置理念。

串并结构

串并结构控制是 TASKCTL 流程控制主要控制策略，同时串并结构也是其它很多功能应用设计的基础，与其它特征应用结合使用会发挥更有效的功能。

对于串并组节点与其它功能的结合在本节不讨论，本节只讨论单纯的串并节点应用。

serial -串行

串行组节点通过 serial 表示，表示在串行组内，所有子节点都依次执行。以下是一个串行设计：

```
7  <serial>
8    <name>g_serial1</name>
9    <sh>
10     <name>shjob</name>
11     <progname>$workhome/bin/comm/trans.sh</progname>
12   </sh>
13  <dsjob>
14    <name>dsjob</name>
15    <progname>VER3CCBExchangeRateToMDB</progname>
16  </dsjob>
17  <python>
18    <name>pyjob</name>
19   <progname>$workhome/src/comm/python/mdb/app/execarysql.py</progname>
20  </python>
21 </serial>
```

上图中，在串行组节点 g_serial1 下分别定义了名称为 shjob、dsjob、pyjob 三个不同种类的作业，三个作业是相互依赖的关系，即它们的运行关系是串行依次执行的关系。

parallel-并行

并行组节点通过 parallel 表示，表示在并行组内，所有子节点相互之间不存在运行先后关系，都可同时执行。以下是一个并行设计：

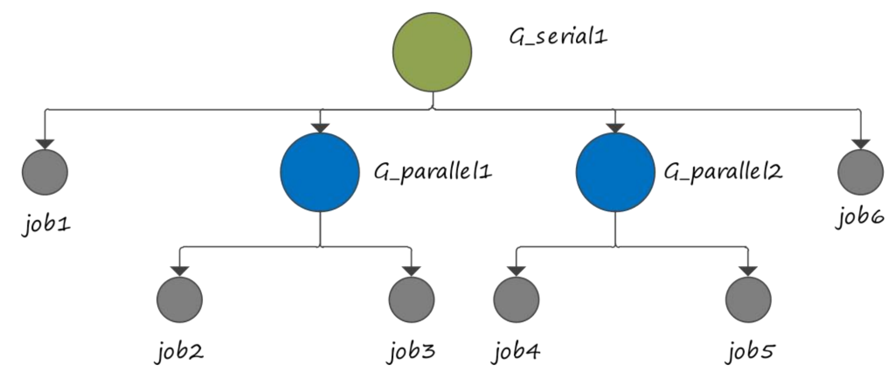
```
13 <parallel>
14   <name>g_parallel</name>
15   <sh>
16     <name>shjob2</name>
17     <progname>$workhome/bin/comm/trans.sh</progname>
18   </sh>
19   <dsjob>
20     <name>dsjob2</name>
21     <progname>VER3CCBExchangeRateToMDB</progname>
22   </dsjob>
23   <python>
24     <name>pyjob2</name>
25     <progname>$workhome/src/comm/python/mdb/app/execarysql.py</progname>
26   </python>
27 </parallel>
```

上图中，在并行组节点 g_parallel 下分别定义了名称为 shjob2、dsjob2、pyjob2三个不同种类的作业，三个作业是不存在相互依赖的关系，它们均可同时运行。

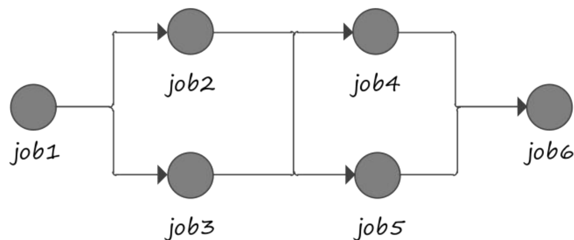
串并嵌套

在实际应用中，简单的串并是无法满足复杂作业依赖关系，只有灵活应用串并嵌套才可能完成。由于排版原因，我们以下以示意图方式对嵌套进行讲解：

一个简单的串并嵌套示意图如下：



上图是相关代码示意图，描述了在 G_serial1 串行节点内依次定义作业节点 job1、并行节点G_parallel1、并行节点 G_parallel2 以及 job6，同时，在并行节点G_parallel1 与 G_parallel2 下分别定义作业节点 job2、job3 与 job4、job5。该示意图实际运行关系如下图所示：



在实际应用中，不论串并节点数还是嵌套深度都远比以上复杂。对于复杂的应用在此不作一一描述，具体需要用户在实际应用中慢慢体会与理解。

循环结构

循环指一个作业可以依次成功运行用户设计的次数。它主要通过作业 cycleinterval 属性来确定，该值缺省是 1，表示只能成功运行 1 次，即无循环。

以下是一个作业的循环设计：（语法有误）——cycle属性，语法变更为cycleinterval

```
14 <python>
15   <name>pyjob2</name>
16   <cycle>10</cycle>  → 表示成功循环运行10次
17   <progname>$workhome/src/comm/python/mdb/app/execarysql.py</progname>
18   <para>$cycle</para>
19 </python>
20
```

以上设计表示 pyjob2 作业需要成功运行 10 次才表示该作业调度完成。一般情况，循环使用会涉及到缺省参数 cycleinterval的应用，每次运行该值都会随着成功次数的改变而改变。该值从 1 开始，成功一次后，该值变为 2，以此类推。通过 cycle 参数带入实际程序中，让实际程序根据cycleinterval值做实际意义的循环处理。

如果要对一个作业流分支进行循环控制，v7.0+版本可以通过把作业流分支组织为include模块节点。然后在include节点上应用cyclecount循环次数和cyclebreak循环中断条件来控制。

```
14 <include>
15   <name>SubModul2</name>
16   <cyclecount>10</cyclecount>
17   <cyclebreak>getjstate ('job2')!=40</cyclebreak>
18 </include>
```

条件分支结构

条件分支功能首先是建立在串并组节点基础之上，一个分支表示一个串并组，同时，条件分支需要借助节点 condition 属性来协同完成。对于 condition 属性，是一个相对复杂的属性，该属性的具体信息在自定义控制策略章节会详细描述，但通过本节，用户可以充分理解什么是 TASKCTL 条件分支。

以下，我们通过一段代码来认识条件分支：

```
7  <serial>
8  <sh>
9    <name>mydeal</name>
10   <progname>$HOME/bin/mydeal.sh</progname>
11 </sh>
12 <!-- 分支mydeal1 串行分支-->
13 <serial>
14   <name>mydeal</name>
15   <condition>if(getjresult("mydeal")==40) CTL_DOIT else CTL_IGN </condition>
16   ...
17 </serial>
18 <!-- 分支mydeal 并行分支-->
19 <parallel>
20   <name>mydeal2</name>
21   <condition>if(getjresult("mydeal")==41) CTL_DOIT else CTL_IGN</condition>
22   ...
23 </parallel>
24 </serial>
```

上图中，在一个大串行组下分别定义一个作业 mydeal 以及 mydeal1 与 mydeal2 两个分支组。如果按串并规则，会先运行 mydeal 作业，再运行 mydeal1 串行组，最后运行 mydeal2 并行组，但由于我们分别在 mydeal1 分支组以及 mydeal2 分支组节点设置了 condition 属性，实际运行就会改变这种简单串行控制策略，这种改变体现在：mydeal1 分支只能在 mydeal 作业运行结果为 40 时运行，而 mydeal2 分支只能在 mydeal 作业运行结果为 41 时运行。

互斥

互斥表示两个不同作业不能同时运行。作业互斥通过作业 ostr 属性实现。例如：

```
14  <parallel>
15    <name>G_GetData</name>
16    <sh>
17      <name>Getdata1</name>
18      <ostr>LockS1</ostr>
19      <progname>$TASKCTLDIR/demo/job/test1.sh</progname>
20    </sh>
21    <serial>
22      <name>G_GetData001</name>
23      <sh>
24        <name>GetData3</name>
25        <ostr>LockS1</ostr>
26        <progname>$TASKCTLDIR/demo/job/errjob.sh</progname>
27      </sh>
28      <sh>
29        <name>GetData4</name>
30        <progname>$TASKCTLDIR/demo/job/test2.sh</progname>
31      </sh>
32    </serial>
33  </parallel>
34
```


在上图中，两个作业 GetData1 与 GetData3 按串并关系，本来是可以同时运行的作业，但由于通过 ostr 属性设置相同互斥资源，从而使它们实现互斥并不能同时运行。

关于互斥资源的设置，主要通过一个自定义字符串来表示，只要名称相同，就表示相关作业互斥。

作业之间的互斥不仅在相同模块中实现，也可以在不同模块甚至不同流程之间体现，只要资源名称相同即可。实际上，在具体应用中，同一流程中距离相近的作业一般不会设置互斥，因为完全可以通过简单串并进行避免两个作业的同时运行，互斥更多的实际应用场景是在不同流程的作业之间实现互斥，这种距离遥

远的作业，采用互斥属性更有实际意义。

另外，互斥只能对部署在同一调度服务器的流程生效，互斥资源名称的异同也是相对同一调度服务器而言。对于不同调度服务器之间的流程不会产生任何关系。

TASKCTL 的 goto 语句 – lean(强制依赖)

前面我们已经讲过，作业之间的先后关系依赖关系，主要通过串行组来实现，但实际应用中，可能会发现两个需要依赖的作业很难通过串并组来实现。对于这种情况，我们可以通过强制依赖属性 lean 来实现。

以下是一段强制依赖设置的代码设计：

```
14  <sh>
15    <name>GetData2</name>
16    <lean>GetData10</lean>
17    <progname>$TASKCTLDIR/demo/job/</progname>
18  </sh>
```

通过 lean 属性设置，表示 GetData2 运行依赖本流程中 GetData10 任务

实际上，TASKCTL 强制依赖很类似一般程序语言中的 goto 语句，我们可以通过 goto 语句在程序内实现自由跳转。但同时我们也知道，一般情况下，语言设计思想都不建议使用或过多使用 goto 语句，而是尽量采用优良的逻辑设计去避免 goto 的使用，以便保证程序的健壮性与可读性。在 TASKCTL 中也是同样的道理，我们不建议用户使用 lean 属性，而是在流程设计时，多分析作业之间的关系，尽量采用串并条件分支等结构化的思路实现相应的功能，这样会使流程代码以及作业关系更清晰、更便于管理。总之，我们要认识到，lean 要尽量少用或不用，它只是 TASKCTL 流程设计思想中结构化控制的补充手段。

提示： 如果要依赖多个作业，可以用,逗号隔开作业名。

执行计划控制策略

执行计划控制策略在调度应用中非常普遍，是调度控制策略中最重要的策略之一。执行计划指作业的运行周期，简单说，指一个作业什么时候需要运行，比如每周一、每月初、每月底以及季末等。

在 TASKCTL 中，执行计划非常灵活，几乎可以定义任意周期，同时，TASKCTL 可以分别支持自然日期执行计划与逻辑日期执行计划。技术上，主要通过 datetype 与 period 两个属性结合使用来完成灵活的执行计划。

datetype-日期类型

datetype 日期类型主要分自然日期与逻辑日期

- ✓ 自然日期：自然日期即系统日期，用 msystime 表示，datetype 缺省就为 msystime，此时通常不需要用户再修改。
- ✓ 逻辑日期：一个流程中可以存在多个逻辑日期，它主要通过私有参数中以类型为 date 参数进行定义，比如一般常用的 workdate 参数。如果我们以逻辑日期确定执行计划时，datetype 设置为相应 date 类型的私有变量参数名称即可。

period-计划表达式

period 计划表达式主要是在 unix 系统的 Crontab 设计思想基础上进行改造，5.0 及以前增加了时间窗口特征（详情参见历史版本代码规则语法文档），但从 5.1 版本开始，taskctl 为了简化用户对 period 的使用难度，精简了时间窗口特征，仅仅保留 [日] [月] [周]。

执行计划表达式格式与说明

[日] [月] [周]

整个表达式由三个字段组成，字段间通过空格分隔。

- ✓ 日：*表示每日；0 表示月末；-1 表示不按日判断，而是由周决定
- ✓ 月：*表示每月

- 周：*表示每天；0-6 分别代码礼拜天到礼拜六

在对日月周设置时，多个数字用’,’隔开，但不能有空格，如果是一段连续数字，可以按如下表达：

比如日字段：1，3-15：表示 1 号，3 至 15 号。

表达式例子

- “* * *”: 表示每天可执行，缺省就是 * * *
- “0 6,12 *”: 表示每年 6 月月末、12 月月末可执行
- “-1 * 1,4”: 表示每周一、四可执行
- “2,4-10 1,2 *”: 表示 1,2 月的 2 号以及 4 到 10 号可执行

执行计划应用案例

以下通过代码举例说明执行计划的应用：

例子 1

```
14  <sh>
15    <name>demo_sh</name>
16    <progname>$TASKCTLDIR/demo/shell/myselfcondition.sh</progname>
17    <period>1,15 1,4,7,9 *</period>
18    <jobdesc>1,4,7,9月1,15号跑</jobdesc>
19  </sh>
```

以上计划按自然日期确定。表示每年 1、4、7、9 月，每 1、15 日可以执行。

例子 2

```
21  <sh>
22    <name>demo_sh1</name>
23    <progname>$TASKCTLDIR/demo/shell/myselfcondition.sh</progname>
24    <period>1,15 1,4,7,9 *</period>
25    <datetype>workdate</datetype>
26    <jobdesc>1,4,7,9月1,15号跑</jobdesc>
27  </sh>
```

以上计划按逻辑日期workdate 参数日期确定。表示每年 1、4、7、9 月，每 1、15 日可以执行。

容错控制策略

容错策略主要表示流程调度运行过程中，作业运行错误后的后续处理机制。容错机制主要有两种方式：错误重做机制与错误忽略机制。

错误重做机制

错误重做机制指作业执行错误后可以根据用户 maxnum 属性设置次数反复重试，直到最大次数为止。如果达到最大次数，该作业还未成功，确定该作业失败，所有依赖该作业的作业都不会执行。

以下通过一段代码说明：

```
14  <sh>
15    <name>job4</name>
16    <progname>$HOME/myshell.sh</progname>
17    <maxnum>3</maxnum>  → 错误最大重调次数
18  </sh>
```

错误忽略机制

错误重做忽略机制指作业执行错误后可以根据 maxnum 属性设置次数反复重试，直到最大次数为止，如果达到最大次数，该作业还未成功，那么通过 ignoreer 属性确定是否忽略错误，上图没有设置，默认为 N，不忽略，当此时显示设置属性值为 Y 时，错误被忽略，所有依赖该作业的作业继续往下执行。

以下通过一段代码说明：

```
14  <sh>
15    <name>job4</name>
16    <progname>$HOME/myshell.sh</progname>
17    <maxnum>3</maxnum>
18    <ignoreerr>Y</ignoreerr>  → 是否忽略错误
19  </sh>
```

循环控制策略

对于一些作业或模块希望循环执行，可以通过设置节点的循环属性来实现。

- ✓ cyclecount（v7.0+） -- 循环次数， v7.0之前版本为 cycle
- ✓ cyclebreak（v7.0+） -- 循环退出条件，满足条件则退回循环模块

✓ cycleinterval（v7.0+） -- 每次循环结束后的等待间隔时间

```
14 <include>
15   <name>SubModul2</name>
16   <cyclecount>10</cyclecount>
17   <cyclebreak>getjstate ("job12")!=40</cyclebreak>
18 </include>
```

返回信息策略

作业的返回信息用来判断该作业调用成功与否。用数字来匹配作业程序的退出码。可使用连串数据：成功返回值 0-10，警告返回值 11-30 等。注意：用户自定义的返回值只能是在 0-100 之间。

v 7.0 + 新增支持返回信息匹配作业程序的输出信息。如：[ORA-]可匹配 oracle 程序的错误特征码。

```
14 <perl>
15   <name>SubModul2_JobNode0</name>
16   <progname>$HOME/myperl.plx</progname>
17   <successv>[SUCCESS]</successv>
18   <failedv>[FATAL]</failedv>
19   <errorv>[ORA-]</errorv>
20   <warningv>[WARNING]</warningv>
21 </perl>
```

✓ successv -- 作业成功状态返回信息

✓ errorv -- 作业错误状态返回信息

✓ failedv -- 作业失败状态返回信息

✓ warningv -- 作业警告状态返回信息

返回信息判断支持两种规则：

✓ 作业程序退出码值规则

✓ 作业程序日志输出信息匹配规则

如果 successv 、errorv 、failedv 、warningv 其中任一返回信息属性应用了“日志输出信息规则”，那么其它返回信息属性应用的“退出码值”规则就无效。

分片作业策略

在一些分布式计算的作业中，通常会用到分片技术，可以通过设置作业的分片执行属性来实现。

- ✓ issplit（v7.0+） -- 是否应用分片执行策略
- ✓ splitcount（v7.0+） -- 分片的个数或分片个数表达式

```
14  <perl>
15    <name>SubModul_JobNode0</name>
16    <issplit>Y</issplit>
17    <splitcount>10</splitcount>
18  </perl>
```

自动执行策略

默认情况下，作业只要满足调度条件后就会自动执行。如果需要对作业进行人为的确认后再执行，那么可以设置 autorun 为“N”。当流程运行到该作业时会暂停。直到进行确认执行后，流程才会继续运行。

```
14  <perl>
15    <name>SubModul_JobNode2</name>
16    <autorun>N</autorun>
17  </perl>
```

优先级策略

作业 priority 属性用于控制并发的优先执行顺序。值越小表示优先级越高。

```
14  <perl>
15    <name>SubModul_JobNode2</name>
16   <priority>99</priority>
17  </perl>
```

取值范围为 1～100，在代码里面预设的 priority 值可以在 Monitor 客户端运行时环境动态调整优先级值。另外，在待执行队列中（作业状态为等待），可以对优先级进行置顶操作。

超时失败策略

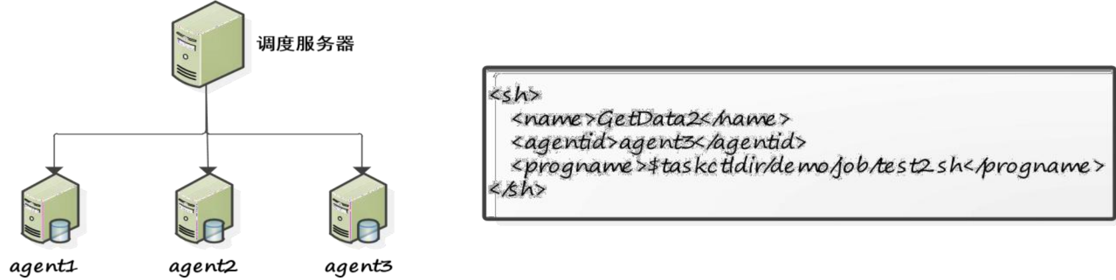
作业 timeout 属性用于控制作业最大运行时间，单位为秒。当作业超时后，状态为失败，流程将暂停执行。当值为 0 时，表示不应用该属性。

```
14  <perl>
15    <name>SubModul_JobNode2</name>
16    <timeout>3600</timeout>
17  </perl>
```

远程调度与负载均衡

运程调度指当作业程序未部署在相应调度服务上时，调度服务器需要通过执行代理控制相应程序。

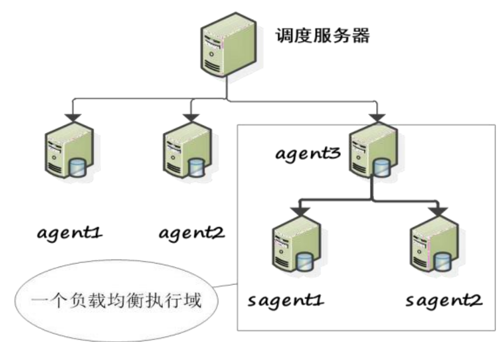
TASKCTL 远程调度比较简单，可以通过作业的 agentid 属性完成，只需通过该属性设置远程程序主机节点所部署的 CTL 执行代理的名称即可。如下图所示：



上图表明调度服务控制 agent3 部署 test2.sh 程序。

对于负载均衡应用，实际上与代码设计无关，只与部署相关。就拿以上调度示例为例，只需在 agent3 下级联从代理并与上级代理做相同的作业程序部署即可完成负载均衡调度。

如下图所示：




```
<sh>
  <name>GetData2</name>
  <agentid>[agent3]</agentid>
  <progname>$taskctldir/demo/job/test2.sh</progname>
</sh>
```

利用 hostuser 实现远程调度

从 v6.0+开始，TASKCTL 为用户提供统一的无代理远程调度机制。相对于代理模式来讲，无代理由于无需在受控目标机器部署相应的软件，即可调度控制相应的作业程序。这种变化，让调度控制空间格局，得到彻底的延展变化，极大拓展了调度的应用场景。这种场景适合运维管理自动化。

请注意，该方式需要配合 ssh 协议来实现。sip 表示远程主机 IP，suser 表示远程主机执行用户，如下图所示：

```
14  <perl>
15    <name>SubModul_JobNode0</name>
16    <progname>$HOME/myperl.plx</progname>
17    <hostuser>sip=192.168.137.11,suser=ods</hostuser>
18  </perl>
```

自定义控制策略

核心调度控制策略的技术本质，就是确定一个作业的执行条件，比如依赖并发等就是属于执行条件之一。其实，调度核心对作业的执行条件远非只有依赖并发这些条件，根据实际经验，我们可以总结出各种执行条件，比如：执行计划、互斥、容错处理等均属于作业的执行条件，但是总结归纳是有限的，变化是无限的，在一些复杂情况下，总会存在一些不可确定的执行条件。为此，我们在众多可总结的条件基础上，增加了用户自定义条件接口，以满足不可确定的调度需求，从而也使 CIR 核心调度体系得以完善。

TASKCTL 自定义控制通过节点 condition 属性完成。该属性功能强大，可以

完成各种复杂的调度应用需求，前面所讲的条件分支就是一个具体例子。

技术方面，condition 主要通过条件表达式来实现，以下通过一段代码来认识 condition 属性的应用：

```
14  <sh>
15      <name>SubModul_JobNode0</name>
16      <progname>ddd</progname>
17      <condition>if(getjresult('job1')==2) CTL_DOIT else CTL_IGN</condition>
18  </sh>
```

if条件表达式

以上代码中，作业 SubModul0_JobNode0 在 job1 作业处理返回结果为 2 时处理，否则，该作业不处理。

condition 条件表达式说明

表达式基本结构

Condition 表达式结构主要是条件表达式结构，其结构如下：

```
if（【布尔运算表达式】）【处理动作1】 else 【处理动作2】
```

由上可知，condition 表达式主要以 if-else 结构为基础，通过布尔运算表达式运算结构决定处理动作，布尔表达式运算结果为真，执行‘处理动作 1’；反之，执行‘处理动作 2’。

布尔运算表达式

表达式局限范围

conditon 中，布尔运算表达式存在一定的限制范围，该表达式只支持 taskctl 提供的内置函数运算，“==”、“!=”、“>=”、“>”、“<”、“<=”、“+”、“-”、“*”、“/”以及“and”、“or”等有限的运算。

内置函数

内置函数主要在运算表达式中使用，所有函数运算结果都返回整数，以便参加布尔运算。主要函数如下表：

分类	函数名	功能	参数	返回值

字符串 处理	substr	截取字符串	参数 1：源字串 参数 2：开始位置（从 0 开始） 参数 3：结束位置	截取后的子串
	addstr	字符串相加	参数 1：源字串 1 参数 2：源字串 2	拼接后的字符串
	strlen	求字串长度	参数 1：源字串	字符串长度数字
	strcmp	字串比较	参数 1：源字串 1 参数 2：源字串 2	非 0：false 0：true
	upper	字串转大写	参数 1：源字串	大写字符串
	lower	字串转小写	参数 1：源字串	小写字符串
时间与 日期相 关处理	systime	按格式获取系统日期时间	参数 1：日期时间格式 说明：yyyy 年，mm 月，dd 日,hh 小时，mi 分, ss 秒	格式化后的时间
	addyear	年加减	参数 1：被加日期时间 参数 2：被加日期时间格式 参数 3：增量（为负值时需用单引号，如'-1'）	修改后的日期时间
	addmonth	月加减		
	addday	天加减		
	addhour	小时加减		
	addminute	分加减		
	addsecond	秒加减		
	formattime	时间格式转换	参数 1：被转换日期时间 参数 2：被转换日期时间格式参数 3：目标转换日期时间格式	格式化后的时间
	dayofweek	判断日期为周几	可以缺省，缺省时为判断系统时间 参数 1：指定日期 参数 2：指定日期格式	返回指定日期为星期几数值 0：星期天，1：星期一，。。。 6：星期六
	isendofmonth	判断日期是否为月末	可以缺省，缺省时为判断系统时间 参数 1：指定日期	0： false 1： true

			参数 2：指定日期格式	
其它	getjresult	获取作业返回值	参数 1：作业名 参数 2：流程名（无表示当前流程）	指定作业的执行结果
	getjstate	获取作业状态		指定作业的处理状态数值，注：详见后续作业状态表
	getjretmsg	获取作业返回消息		作业的处理返回信息
	getvarv	获取变量值		
	execcmd	执行行命令，并获取返回值	系统行命令	系统行命令的exit 值
	getstdout	执行行命令，获取标准输出	系统行命令	行命令标准输出信息
	exp	计算+-*/表达式	表达式	计算结果
	if	if 真假判断	逻辑判断表达式： >、 <、 ==、 <= 、 >= 逻辑与或运算： and、or、in	1: true 0：false

■ getjstate – 作业处理状态表

当前流程指定作业的处理状态值。处理状态值参见下表：

状态值	状态名称	状态说明
-1	NetErr	网络错误
-2	NOCOMMFILE	通信文件不存在
-3	NOPLUGIN	无执行插件
-4	CTLNODEERR	控制接点错

-5	MSGERR	消息队列错
-7	SYSOERR	系统其它错
-8	NOISSUE	程序未发布
-9	ROUTEERR	错误路由
-10	NONETINFO	无网络配置信息
-90	REDO	系统忙
-100	OTHER	未执行
1	INIT	正在执行
2	WAITMANRUN	正在循环
5	SYSBUSS	错误需再次执行
10	RUNNING	错误不再执行
11	CYCLE	成功执行
20	ERR	因周期本期不执行通过
25	BRKERR	忽略错误通过
29	FAILED	该作业因为无效通过
40	SUCCESS	成功执行
41	OKPERIOD	因周期本期(执行计划)不执行通过
42	IGNOREERR	忽略错误(警告)通过
43	DISABLE	设置无效通过
44	FORCEOK	强制通过
45	BRKCYCLE	中断循环通过

处理动作

处理动作，表示当前作业根据布尔表达式的结果进行的处理行为。该处理行为有三种：

- ✓ CTL_DOIT：表示作业执行

- ✓ CTL_IGN：表示作业忽略，不作任何处理
- ✓ CTL_WAIT：表示作业不处理，还继续等待

提示： 在 if 结构中，如果没显示确定 else 处理动作，系统缺省为 CTL_IGN。

condition 属性应用

■ **例子 1：**

```
<condition>if(atol( ‘$(para1)’ ) == 100) CTL_DOIT</condition>
```

说明：

在例子中，我们使用了缺省 else 结构，以及代码变量使用。该例表明如果当前流程变量 para1 的值为 100 时，执行当前作业，否则不执行并忽略通过

■ 例子 2：

```
<condition>if(getjresult( ‘job1’ )) == 10) CTL_DOIT else CTL_IGN</condition>
```

说明：

在例子中，我们使用 if-else 完整结构。该例表明如果 job1 的执行结果是 10 时，执行当前作业，否则不做并忽略通过。

■ 例子 3：

```
<condition>if(execcmd( ‘myexe 1 2’ )) == 5) CTL_DOIT else CTL_WAIT</condition>
```

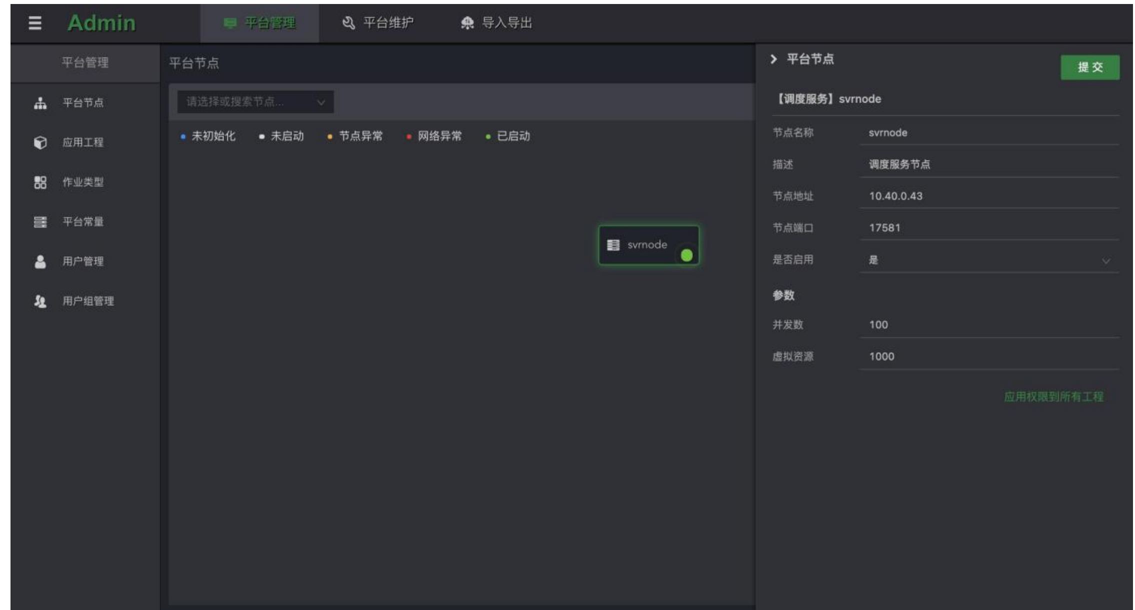
说明：

该例表明如果自定义程序 myexe（并带两个参数 1 与 2）的执行结果是 5 时，执行当前作业，否则继续等待。

虚拟资源控制策略

面对各式各样的调度控制需求，只要利用好上面的控制策略，从逻辑上来说是没有问题的。然而在大量并行作业组，这种实际应用场景中，有些作业仅仅需要几秒钟就能完成，有些作业则可能要花上几个小时时间。不同的作业对系统资源的需求也就不一样。若分配给作业的资源都采用同样的方式，势必会造成资源的浪费。那么要怎样去平衡这样的资源消耗呢？TASKCTL 引入了虚拟资源的概念，通过设置作业的虚拟资源消耗权值，来达到这一目的。

在建立 CTL 代理控制节点的时候，TASKCTL 会为每个代理控制节点默认分配“1000”数值的虚拟资源，最大作业并发数为 100。调度服务端则默认分配“1000”数值的虚拟资源，最大作业并发数为 1000。如下图所示：



假如所有作业都采用系统默认资源消耗值“10”。容许同时并行 10 个作业，第 11 个并行作业则需要等待。只有等这 10 个并行作业中，有作业运行完毕释放部分资源，并满足第 11 个并行作业的资源消耗需求条件，这个作业才会运行(其它控制策略都满足的情况下)。

VirResource 属性的应用：

```
14  <dsjob>
15    <name>MainModul_dsjob1</name>
16    <virresource>100</virresource>
17  </dsjob>
```

说明：设置作业的 virresource 属性，即可调整该作业的虚拟资源消耗值。请注意它只是一个权值，并没有具体的计量单位。且必须小于当前 CTL 控制节点的虚拟资源总值。通常情况下，我们并不需要调整该属性。只有在大规模并行应用的条件下，可通过监控作业一段时期的耗时情况，才根据实际情况对其进行优化调整。

定时控制策略

从技术角度来说，定时控制策略和结构化控制策略中的串行、循环、依赖、互斥是对立的概念。在定时容器中，各个作业的关系都是并列且无序的，这意味着设置作业之间的关系都是无效的。只需要按照一定的时间间隔执行即可。

定时控制策略只在定时控制容器中有效。因此，我们需要新建定时器控制容器。建立好后查看模块代码如下：

```
4        流程名称: project1_Flow1
5        模块名称: MainModul
6   *******************************************************************-->
7   <serial>
8     <name>MainModul_rootnode</name>
9     <begin>
10      <name>MainModul_beginjob</name>
11      <jobdesc>begin</jobdesc>
12    </begin>
13  <sh>
14    <name>MainModul_JobNode0</name>
15  </sh>
16  <parallel>
17   <monititle>第一组</monititle>
18   <name>MainModul_ParallelNode0</name>
19   <timingplan>23010 s 30</timingplan>
20   <sh>
21     <name>MainModul_JobNode1</name>
22   </sh>
23   <sh>
24     <name>MainModul_JobNode2</name>
25   </sh>
26  </parallel>
27  <sh>
28    <monititle>第二组</monititle>
29    <timingplan>* m 10</timingplan>
30    <name>MainModul_JobNode3</name>
31  </sh>
32    <!--  用户模块代码自定义区开始 -->
33    <!--  用户模块代码自定义区结束 -->
34    <end>
35      <name>MainModul_endjob</name>
36      <jobdesc>end</jobdesc>
37    </end>
38  </serial>
```

- Timingplan 定时表达式格式与说明

  [*|hhmiss] [d|h|m|s] [num]

  整个表达式由三部分组成，每部分用空格隔开。

  - ✓ 第二部分的 d|h|m|s 表示时间间隔单位，分别是日，小时，分，秒。只能是其中之一。
  - ✓ 第三部分的 num 表示时间间隔的整数值。
  - ✓ 第一部分的* 和 hhmiss 是可选项，其中*表示控制器启动后立即开始定时控制。“hhmiss”表示开始定时控制的相对时间点。只有通过第二部分和第三部分计算出来的时间间隔，与当前系统时间正负差值的倍数，恰好能满足这个时间点才开始执行。

- **定时表达式例子**
  - ✓ * m 10 表示控制器启动后每隔 10 分钟执行一次
  - ✓ 230101 s 30 表示控制器启动后，当前系统时间的秒部分为 31 或 01 开始，每隔 30 秒钟执行一次
  - ✓ 230101 m 12 表示控制器启动后，若 (当前系统时间 +/- 230101) % 12m = 0s，即当前系统时间与 230101 的正负差值跟 12 分钟的取余结果等于 0 秒，则开始每隔 12 分钟执行一次

Timingplan 属性也支持继承和缺省，若不设置该属性或该属性值为空，则不会定时执行（可以手动执行）。为了能更好的应用定时器，我们可能会用到定时器的 monititle 属性，请参照 5.3.4.3 章节。

5.3.3 流程结构化管理

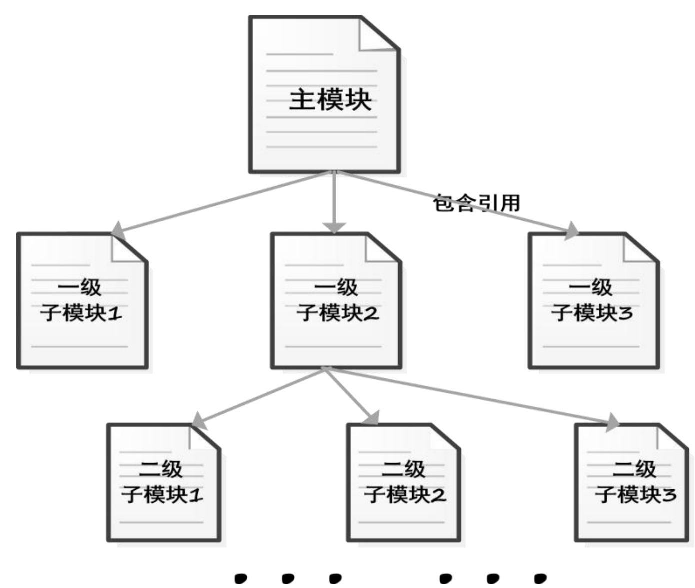
流程结构化管理对调度控制本身不起任何作用，它不能改变任何调度行为，但对流程的应用管理却起到重要的作用。通过流程结构化管理，会使流程更清晰、更容易维护。

流程结构化管理主要通过模块引用以及子流程调用来实现。本节主要讲述怎

么通过代码来实现对流程的结构化管理。

Include – 模块引用

在‘流程结构章节’我们就提到模块的概念，模块是流程核心信息的最基本单位。面对大流程，我们通常会根据业务特征以及 ETL 流程的数据流纵向特征划分模块。这些模块存在一定的级联关系。如下图所示：



由图可知，这种关系主要通过上下级关系串联起来的，这种级联关系在代码中主要通过平台缺省特殊作业节点 include 实现。代码实现如下：

```
14 <include>
15   <name>SubModul1</name>   SubModul1为被包含引用的模块名称
16 </include>
```

如上图代码所示，在上级模块中需要引用的地方，增加 include 节点，并将该节点名称设置需要引用的模块名称即可。

提示： include 节点可应用循环作业分支，也可应用继承属性

flow - 子流程调用

子流程调用，也是流程结构化管理的范畴。关于什么是子流程以及与模块的区别前面“3.3 流程基本属性”章节已经讲过，在此就不再论述，本节主要讲怎么在代码中引用子流程。

在模块代码中，引用子流程主要通过平台缺省特殊作业节点 flow 来实现。代码实现如下：

```
14  <flow>
15    <name>subflow1</name>
16    <progname>subflow1</progname>   → subflow1为被调用的子流程名称
17  </flow>
```

由以上代码可知，子流程的引用方式与模块有一定区别，模块引用直接通过节点 name 名称属性引用，而子流程引用是通过 progname 属性引用。

一般情况下，我们很少使用子流程，但由于子流程与模块的一些本质区别，决定了子流程的一些特殊应用场景。这些应用场景在“流程典型应用案例”章节会作相关描述。

6 典型应用案例

通过以上章节，我们具备了控制器以及模块代码的设计能力，本章节主要通过一些典型案例进一步加深对控制容器以及模块代码的理解。（更多的应用范例请参考 TASKCTL 安装时自带的范例工程）

6.1 流程每个批次的开始触发

作为调度，流程的触发是非常普遍的应用场景。而流程触发最常用的是时间定时触发以及文件到达触发：

时间触发

代码示例如下：

```
7   <serial>
8     <name>MainModul_rootnode</name>
9     <begin>
10      <name>MainModul_beginjob</name>
11      <jobdesc>开始</jobdesc>
12    </begin>
13  <!--
14        第一步：流程触发控制
15  -->
16  <nulljob>
17    <name>startctljob</name>
18    <condition>if($(startdatetime)&lt;systime('yyyymmddhhmiss')) CTL_DOIT else CTL_WAIT</condition>
19    <jobdesc>启动触发控制节点</jobdesc>
20  </nulljob>
21  <!--
22        第二步：调实际业务流程
23  -->
24  <flow>
25    <name>callbussflow</name>
26    <progname>Demo_zbussflow</progname>
27    <jobdesc>调实际业务处理流程</jobdesc>
28  </flow>
```

在流程开始位置，添加了一个 nulljob 空作业，并设置了时间运行条件。只要当时间运行条件满足后，才能执行后续作业，从而达到定时触发的目的。

在实际应用中，对于定时触发流程我们一般需要增加开始运行时间参数 $(startdatetime)，当流程跑完一个批次后，让这个参数值加 1 天，这样它就不会一直小于系统当前时间，以保证流程每天只能运行一次。对于 1 天需要多次运行且又有时间触发的流程，我们会在 “6.4 子流程应用” 章节进行描述。

文件到达触发

文件到达触发是指相关数据文件到达后，指定流程就开始调度运行。平台自身提供了 filewatch 文件到达作业类型，通过简单的配置它的 para 属性（其值为文件路径），就能快速方便的监控文件是否到达。

以下是文件达到触发例子。

```
18  <!-- 用户代码自定义区域开始 -->
19  <!--
20      在流程开始，通过filewatch任务类型判断一个相应文件是否达到，而且该文件名
21      带有日期标志
22  -->
23  <filewatch>
24    <name>startctljob</name>
25    <para>/home/taskctl/ods/$(workdate)/fileisok.fig</para>
26    <jobdesc>文件达到判断</jobdesc>
27  </filewatch>
28  <!--
29    调实际业务流程
30  -->
31  <flow>
32    <name>callbussflow</name>
33    <progname>Demo_zbussflow</progname>
34    <jobdesc>调实际业务处理流程</jobdesc>
35  </flow>
```

在流程文件到达触发实际应用中，文件到达一般是相对业务日期而言，即哪个业务日期的文件到达，因此在判断文件是否到达程序中，一般需要一个业务日期参数。如上图所示$(workdate)参数的运用。

6.2 流程翻牌处理

流程翻牌是 ETL 流程中一个常用概念，简单地说，流程翻牌表示一个批次流程结束，同时可能会修改一些与业务相关的信息，比如业务日期，处理一个批次后，需要将日期修改为下一天。

代码示例如下：

```
18 <nulljob>
19   <name>startctljob</name>
20   <condition>if($(startdatetime)&lt;systime('yyyymmddhhmiss')) CTL_DOIT else CTL_WAIT</condition>
21   <jobdesc>启动触发控制节点</jobdesc>
22 </nulljob>
23 <!--
24     第二步：调实际业务流程
25 -->
26 <flow>
27   <name>callbussflow</name>
28   <progname>Demo_zbussflow</progname>
29   <jobdesc>调实际业务处理流程</jobdesc>
30 </flow>
31 <!--
32     翻牌修改变量
33 -->
34 <modivarv>
35   <name>passflow</name>
36   <para>varname=startdatetime,varvalue=$(addday($(startdatetime),'yyyymmddhhmiss',1))</para>
37   <jobdesc>翻牌：修改变量</jobdesc>
38 </modivarv>
39   <end>
40     <name>MainModul_endjob</name>
41     <jobdesc>结束</jobdesc>
42   </end>
43 </serial>
```

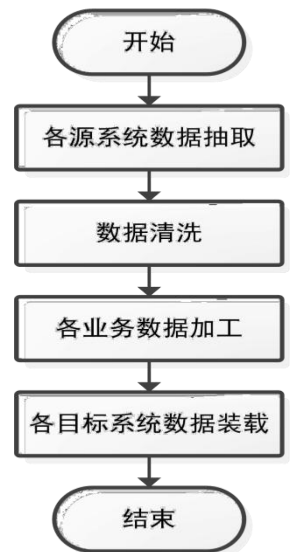
与开始触发类似，平台自身不提供流程翻牌机制。流程的核心只是单纯的对作业的调度管理。这种具有一定业务特征的过程都留给客户通过作业自行完成。

在上述例子中，在流程最后，我们通过 modivarv 作业修改时间变量 $(startdatetime) 增加为下一天来完成翻牌处理。关于“addday”及更多函数的运用，可参考 TASKCTL 安装时自带的的范例（“所有内置函数的使用”）。

6.3 流程多模块设计

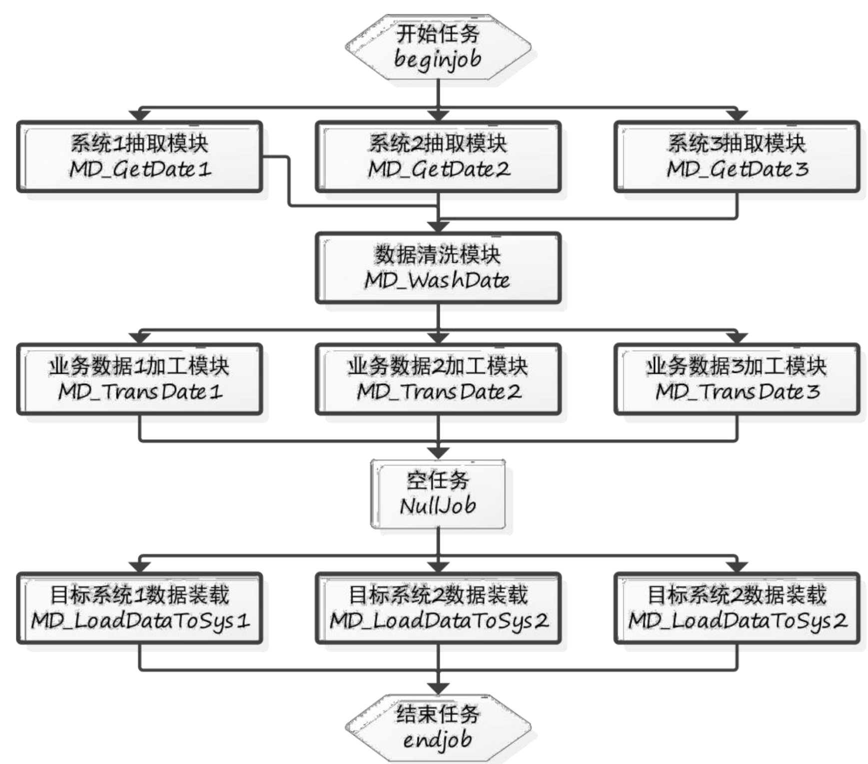
流程多模块设计是 TASKCTL 流程设计的核心技术之一，也是最为广泛的应用。在本节点，我们主要从一个实际 ETL 应用需求出发，并通过多模块技术来设计流程。

一个ETL 流程需求示例



流程实现

对以上流程需求，我们采用一个多模块方式实现。主要是将需求中每部分通过一个模块或多个模块实现，实现流程如下图所示：



以上各个子模块由项目相关成员分别完成，然后在流程主模块中分别定义 include 节点引用相关模块即可。主模块代码设计如下：

```
7    <serial>
8      <name>MyETLFlow</name>
9      <begin>
10         <name>beginjob</name>
11     </begin>
12     <!-- 各源系统数据抽取 -->
13    <parallel>
14      <name>MainModul_ParallelNode0</name>
15      <include>
16        <name>MD_Getdate1</name>
17      </include>
18      <include>
19        <name>MD_Getdate2</name>
20      </include>
21      <include>
22        <name>MD_Getdate3</name>
23      </include>
24    </parallel>
25     <!--  数据清洗 -->
26    <include>
27        <name>MD_Washdate</name>
28      </include>
29     <!--  各业务数据加工 -->
30
31   <parallel>
32       <name>MainModul_ParallelNode1</name>
33       <include>
34         <name>MD_TransDate1</name>
35       </include>
36       <include>
37         <name>MD_TransDate2</name>
38       </include>
39       <include>
40         <name>MD_TransDate3</name>
41       </include>
42     </parallel>
43
44      <nulljob>
45        <name>nulljob</name>
46      </nulljob>
47      <!--  各目标系统数据转载 -->
48    <parallel>
49       <name>MainModul_ParallelNode2</name>
50       <include>
51         <name>MD_LoadDataToSys1</name>
52       </include>
53       <include>
54         <name>MD_LoadDataToSys2</name>
55       </include>
56       <include>
57         <name>MD_LoadDataToSys3</name>
58       </include>
59     </parallel>
60      <end>
61        <name>MainModul_endjob</name>
62      </end>
```

6.4 子流程应用

以下以一个每天需要处理多个批次的流程来说明子流程的应用， 该流程每天早中晚分别处理， 业务日期即自然日期，且每天第三批次处理完后做特殊翻牌处理。

TASKCTL 对以上需求处理如下：

设计子流程

该子流程即每天多批次处理的实体流程。我们将该流程命名为 subflow，对于子流程代码在此不作介绍，它的设计与一般流程一样。

编写特殊翻牌作业

翻牌作业假如通过 shell 程序实现，并命名为 passday.sh，具体内容在此也不做介绍。

调用定时器设计

调用定时器通过 flow 节点调用子流程，代码如下：

```
7    <serial>
8      <name>MainModul_rootnode</name>
9      <begin>
10       <name>MainModul_beginjob</name>
11     </begin>
12     <!--  用户模块代码自定义区开始 -->
13   <flow>
14     <name>no1</name>
15     <progname>subflow</progname>
16     <timingplan>080000 d 1</timingplan>
17   </flow>
18     <!--  每天8点执行subflow -->
19   <flow>
20     <name>no2</name>
21     <progname>subflow</progname>
22     <timingplan>120000 d 1</timingplan>
23   </flow>
24     <!--  每天12点执行subflow -->
25   <flow>
26     <name>no3</name>
27     <progname>subflow</progname>
28     <timingplan>180000 d 1</timingplan>
29   </flow>
30     <!--  每天18点执行subflow -->
31     <!--  用户模块代码自定义区结束 -->
32     <end>
33       <name>MainModul_endjob</name>
34     </end>
35   </serial>
```

6.5 回算流程

在介绍回算流程之间，我们首先要明白什么是回算。在实际正常的 ETL 处理过程中，每天都按业务日期完成一系列抽数、清晰、计算等 ETL 处理过程。但有时因某些数据的人为变化，我们需要从过去的某个业务日期开始，一天一天地重新处理某部分作业作业，直到当前业务日期为止。对这种计算，我们称之为回算。

回算流程可以说是调度中最复杂的流程之一，调度要自动完成这个回算流程，要面临很大的挑战。许多调度都回避这个问题，由客户自己处理或者回算时由人工处理。

在 TASKCTL 调度平台中，充分利用各种控制策略，可以由调度实现回算流程。在回算应用案例中，分别使用了 Condition、循环、子流程等技术。

编写回算判断作业

该作业程序用 c 编写完成，它需要使用调度核心控制接口。该程序根据一些业务数据，判断出是否需要回算，并决定回算的起始日期以及回算循环次数（即回算到当前业务日期的批次数），并将这些信息通过接口告诉调度平台。

该程序在编写时需确定回算与否的返回值约定。

在此，我们假设该程序名称为 preback。

配置回算流程

该流程主要在回算需要处理作业的流程。在此，我们假设该流程的名称为 backflow。

配置主流程

主流程即正常处理流程。通过该流程的判断决定是否回算。配置如下：

```
7   <serial>
8     <name>mymain</name>
9     <begin>
10      <name>MainModul_beginjob</name>
11      <jobdesc>beginjob</jobdesc>
12    </begin>
13  <parallel>
14    <name>flowcore</name>
15    ...
16    <serial>
17     <!--
18        1.preback为回算处理判断处理流程，它返回40表示需要回算
19        2.在该程序我们需要计算回算次数以及起始日期。其中回算次数即循环次数，通过接口设置
20        backflowjob任务的cycle属性；例外，将起始日期通过接口设置为backflow流程workdate
21        变量backflowjob任务及backflow流程
22     -->
23      <exe>
24        <name>preback</name>
25        <progname>preback</progname>
26      </exe>
27      <!--backflow流程为回算处理流程-->
28      <flow>
29        <name>backflowjob</name>
30        <condition>if(getjresult("preback")==40) CTL_DOIT else CTL_IGN</condition>
31        <progname></progname>
32      </flow>
33    </serial>
34    ...
35  </parallel>
36    <end>
37      <name>MainModul_endjob</name>
38      <jobdesc>endjob</jobdesc>
39    </end>
40  </serial>
```